Chapter 4 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.

4 Actor Libraries

Authors: *Elaine Cheong*
Christopher Hylands
Edward A. Lee
Steve Neuendorffer
Yuhong Xiong

Contributors: *Chamberlain Fong*
Mudit Goel
Bart Kienhuis
Edward A. Lee
Michael Leung
Jie Liu
Xiaojun Liu
Sarah Packman
Shankar Rao
Michael Shilman
Jeff Tsay
Brian K. Vogel
Paul Whitaker

4.1 Overview

Ptolemy II focuses on component-based design. In this design approach, components are aggregated and connected to construct a model. One of the advantages of component-based design is that reuse of components becomes possible. Polymorphism is one of the key tenets of object-oriented design. It refers to the ability of a component to adapt in a controlled way to the type of data being supplied. For example, an addition operation is realized differently when adding vectors than when adding scalars. In Ptolemy II, use of polymorphism maximizes the potential for reuse of components.

We call this classical form of polymorphism *data polymorphism*, because components are polymorphic with respect to data types. A second form of polymorphism, introduced in Ptolemy II, is *domain polymorphism*, where a component adapts in a controlled way to the protocols that are used to exchange data between components. For example, an addition operation can accept input data delivered by any of a number of mechanisms, including discrete events, rendezvous, and asynchronous message passing.

Ptolemy II includes libraries of polymorphic actors that use both kinds of polymorphism to maximize reusability. Actors from these libraries can be used in a broad range of domains, where the domain provides the communication protocol between actors. In addition, most of these actors are data polymorphic, meaning that they can operate on a broad range of data types. In general, writing data and domain polymorphic actors is considerably more difficult than writing more specialized actors. This chapter discusses some of the issues.

4.2 Actor Classes

Figure 4.1 shows a UML static structure diagram for the key classes in the ptolemy.actor.lib package (see appendix A of chapter 1 for an introduction to UML). All the classes in figure 4.1 extend TypedAtomicActor, except TimedActor and SequenceActor, which are interfaces. TypedAtomicActor is in the ptolemy.actor package, and is described in more detail in volume 2, on software architecture. For our purposes here, it is sufficient to know that TypedAtomicActor provides a base class for actors with ports where the ports carry typed data (encapsulated in objects called *tokens*).

None of the classes in figure 4.1 have any methods, except those inherited from the base classes (which are not shown). The classes in figure 4.1 do, however, have public members, most of which are instances of TypedIOPort or Parameter. By convention, actors in Ptolemy II expose their ports and parameters as public members, and much of the functionality of an actor is accessed through its ports and parameters.

Many of the actors are *transformers*, which extend the Transformer base class. These actors read input data, modify it in some way, and produce output data. Some other actors that also have this character, such as AddSubtract, MultiplyDivide, and Expression, do not extend Transformer because they have somewhat unconventional port names. These actors are represented in figure 4.1 by the box labeled "... Other Actors ...".

The stacked boxes labeled "... Transformers ..." and "... Other Actors ..." in figure 4.1 are not standard UML. They are used here to refer to a set of actors that are listed below. There are too many actors to show them individually in the static structure diagram. The diagram would lose its utility because of the resulting clutter.

Most of the library actors can be used in any domain. Some domains, however, can only execute a subset of the actors in this library. For example, the CT (continuous time) domain, which solves ordinary differential equations, may present data to actors that represent arbitrarily spaced samples of a continuous-time signal. For such signals, the data presented to an actor cannot be assumed by the actor to be a sequence, since the domain determines how closely spaced the samples are. For example, the SampleDelay actor would produce unpredictable results, since the spacing of samples is likely to be uneven over time.

The TimedActor and SequenceActor interfaces are intended to declare assumptions that the actor makes about the inputs. They are empty interfaces (i.e., they contain no methods), and hence they are used only as markers. An actor that implements SequenceActor declares that it assumes its inputs are sequences of distinct data values, and that it will produce sequences of distinct data values as outputs. In particular, the input must not be a continuous-time waveform. Thus, any actor that will not work properly in the CT domain should declare that it implements this interface[1]. Most actors do not implement SequenceActor, because they do not care whether the input is a sequence.

An actor that implements the TimedActor interface declares that the current time in a model execution affects its behavior. Currently, all domains can execute actors that implement TimedActor, because all directors provide a notion of current time. However, the results may not be what is expected. The SDF (synchronous dataflow) domain, for example, does not advance current time. Thus,

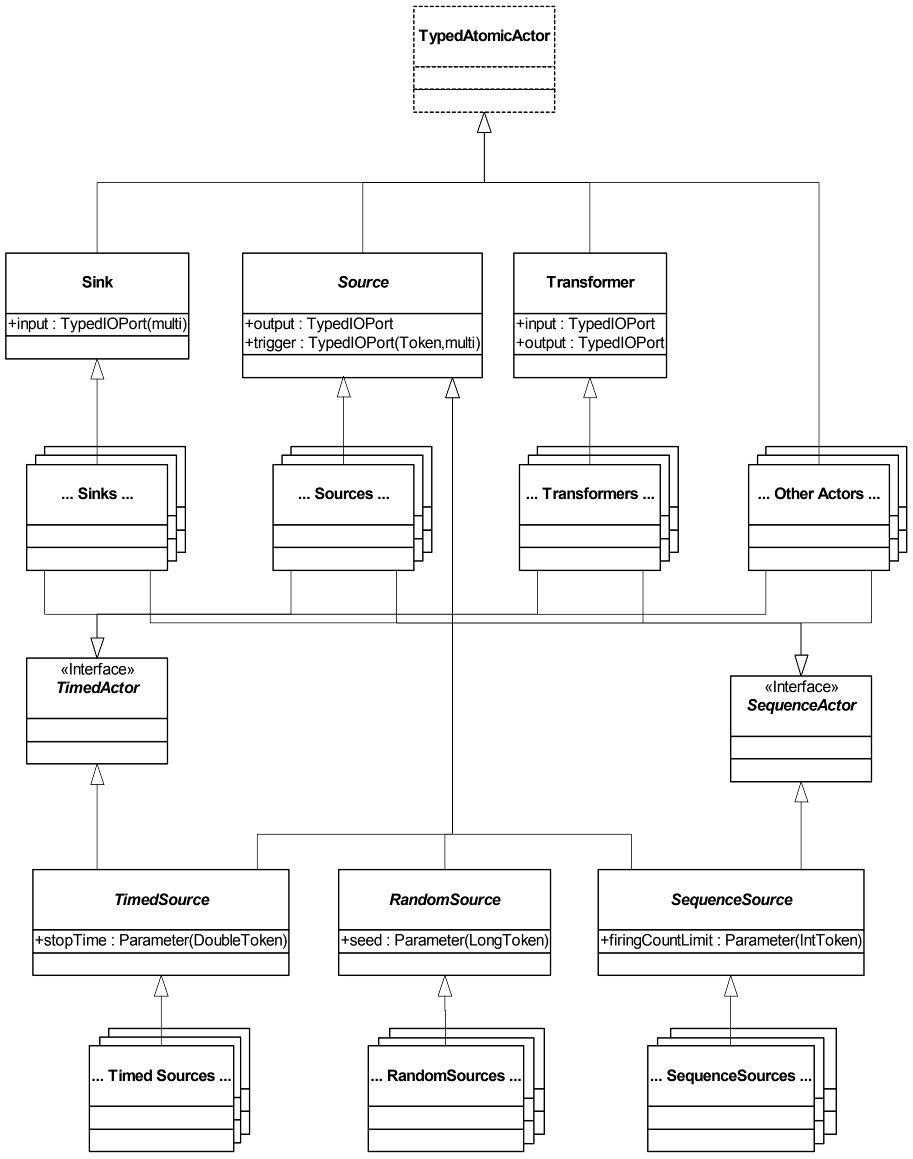

FIGURE 4.1. Key actor base classes and interfaces.

1. Unfortunately, a scan of the current actor library (as of version 4.0) will reveal that we have not been very rigorous about this, and many actors that make a sequential assumption about the input fail to implement this interface. We are working on a more rigorous way of making this distinction, based on the concept of behavioral types.

if SDF is the top-level domain, the current time will always be zero, which is likely to lead to some confusion with timed actors.

4.3 Actor Summaries

In this section, we summarize the actors that are provided in the default Vergil actor library, shown at the left-hand side of the window in figure 4.2. Note that this library is organized for user convenience, and the organization does not exactly match the package structure. Here, we give brief descriptions of each actor to give a high-level view of what actors are available in the library. Refer to the class documentation for a complete description of these actors (in Vergil, you can right-click on an icon and select “Get Documentation” to get the detailed documentation for an actor).

It is useful to know some general patterns of behavior:

- Unless otherwise stated, actors will read at most one input token from each input channel of each input port, and will produce at most one output token. No output token is produced unless there are input tokens.
- Unless otherwise stated, actors can operate in all domains except the FSM (finite state machine) domain, where components are instances of the State class. Additionally, actors that implement the SequenceActor or TimedActor interfaces may be rejected by some domains.

4.3.1 Sources

A source actor is a source of tokens. Most source actors extend the Source base class, as shown in figure 4.1. Such actors have a *trigger* input port, which in some domains serves to stimulate an output. In the DE (discrete event) domain, for example, an input at the *trigger* port causes the current value of the source to be produced at the time stamp of the trigger input. The *trigger* port is a multiport, mean-

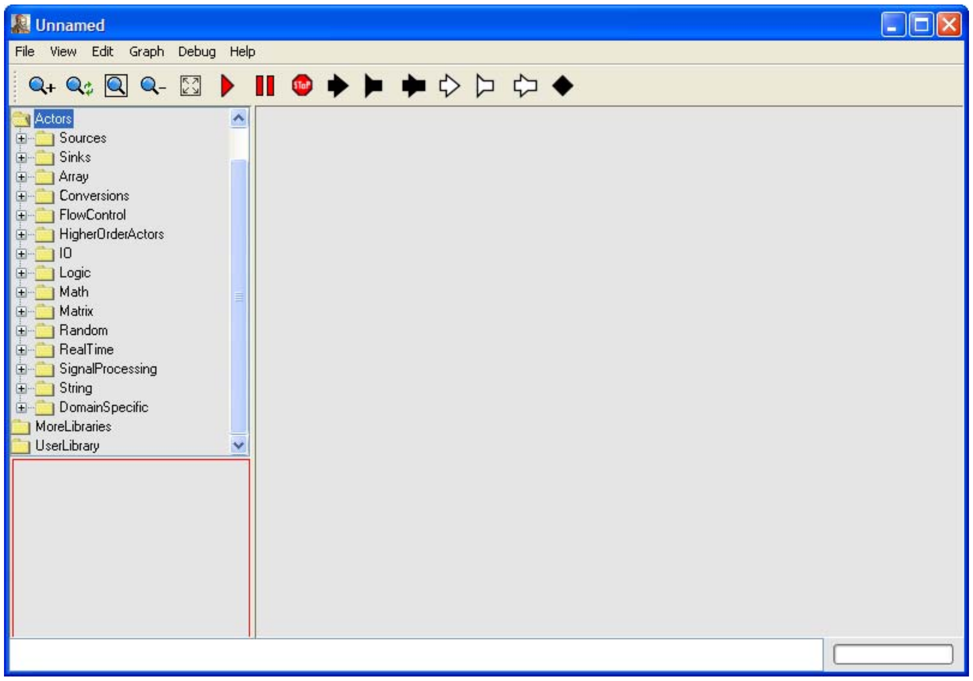

FIGURE 4.2. The default Vergil actor library is shown at the left, expanded to the first level.

ing that multiple channels can be connected to it. The *trigger* port can also be left unconnected in domains that will invoke the actor automatically (SDF, DT, PN, ...). There is no need for a trigger in these domains.

Some source actors use the fireAt() method of the director to request that the actor be fired at particular times in the future. In domains that do not ignore fireAt(), such as DE, such actors will fire repeatedly even if there is no trigger input. In the DE domain, the fireAt() method schedules an event in the future to refire the actor.

Source actors that extend TimedSource have a parameter called *stopTime*. When the current time of the model reaches this time, then the actor requests of the director that this actor not be invoked again. Thus, *stopTime* can be used to generate a finite source signal. By default, the *stopTime* parameter has value 0.0, which indicates unbounded execution.

Source actors that extend SequenceSource have a parameter called *firingCountLimit*. When the number of iterations of the actor reaches this limit, then the actor requests of the director that this actor not be invoked again. Thus, *firingCountLimit* can be used to generate a finite source signal. By default, the *firingCountLimit* parameter has value 0, which indicates unbounded execution.

In some domains, such as SDF and DT, it makes no sense to stop the execution of a single actor. The statically constructed schedule would be disrupted. In these domains, when the specified *stopTime* or *firingCountLimit* is reached, the execution of the entire model will stop.

Some of the most useful actors are *Clock*, which is used extensively in DE models to trigger regularly timed events; *Ramp*, which produces a counting sequence; *Const*, which produces a constant; and *Pulse*, which produces an arbitrary sequence. In Vergil, the source library is divided into *generic sources*, *timed sources*, and *sequence sources*. The first group includes only one source, *Const*, which is agnostic about whether its output is interpreted as a timed output or a sequence output. The other two groups contain actors for which the output is either timed or is logically a sequence.

Generic Sources

Const (extends *Source*): Produce a constant output with value given by the *value* parameter.

Timed Sources

Clock (extends *TimedSource*): Produce samples of a piecewise constant signal with the specified values. The transitions between values occur with the specified period and offsets within the period. This actor uses fireAt() to schedule firings when time matches the transition times, and thus will at least produce outputs at these times. To generate a continuous-time clock, you will likely want to use ContinuousClock instead; that version produces two outputs at the transition times, one with the old value and one with the new.

CurrentTime (extends *TimedSource*): Produce an output token with value equal to the current time (the model time when the actor is fired).

PoissonClock (extends *TimedSource*): Produce samples of a piecewise constant signal with the specified values. The transitions between values occur according to a Poisson process (which has random interarrival times with an exponential distribution). This actor uses fireAt() to schedule firings when time matches the transition times, and thus will at least produce outputs at these times.

TimedSinewave (composite actor) Output samples of a sinusoidal waveform taken at *current time* (when the actor is fired). Note that to generate a continuous-time sine wave in the CT domain you probably want to use *ContinuousSinewave* instead.

TriggeredClock (extends *Clock*): This actor is an extension of Clock with a *start* and *stop* input. A token at the *start* input will start the clock. A token at the *stop* input will stop the clock, if it is still running. To generate a continuous-time clock, you will likely want to use *TriggeredContinuousClock* instead; that version produces *two* outputs at the transition times, one with the old value and one with the new.

VariableClock (extends *Clock*): An extension of Clock with an input to dynamically control the period. NOTE: This actor will likely be replaced at some point by a version of Clock with a *period* PortParameter.

Sequence Sources

InteractiveShell (extends *TypedAtomicActor*): This actor creates a command shell on the screen, sending commands that are typed by the user to its output port, and reporting strings received at its input by displaying them. Each time it fires, it reads the input, displays it, then displays a command prompt (which by default is ">>"), and waits for a command to be typed. The command is terminated by an enter or return character, which then results in the command being produced on the output.

Interpolator (extends *SequenceSource*): Produce an output sequence by interpolating a specified set of values. This can be used to generate complex, smooth waveforms.

Pulse (extends *SequenceSource*): Produce a sequence of values at specified iteration indexes. The sequence repeats itself when the *repeat* parameter is set to true. This is similar to the Clock actor, but it is not timed. Whenever it is fired, it progresses to the next value in the *values* array, irrespective of the current time.

Ramp (extends *SequenceSource*): Produce a sequence that begins with the value given by *init* and is incremented by *step* after each iteration. The types of *init* and *step* are required to support addition.

Sinewave (composite actor): Output successive samples of a sinusoidal waveform. This is a sequence actor. The timed and continuous versions are *TimedSinewave* and *ContinuousSinewave* respectively.

SketchedSource (implements *SequenceActor*): Output a signal that has been sketched by the user on the screen.

4.3.2 Sinks

Sink actors are the ultimate destinations for tokens. Sink actors have no outputs, and include actors that display data in plots, textual form, or tables.

Many of these actors are shown in figure 4.3, which shows a UML static structure diagram. Several of these sinks have both time-based and sequence-based versions. TimedPlotter, for example, displays a plot of its input data as a function of time. SequencePlotter, by contrast, ignores current time, and uses for the horizontal axis the position of an input token in a sequence of inputs. XYPlotter, on the other hand, uses neither time nor sequence number, and therefore implements neither TimedActor nor SequenceActor. All three are derived from Plotter, a base class with a public member, *plot*, which implements the plot. This base class has a *fillOnWrapup* parameter, which has a boolean value. If the

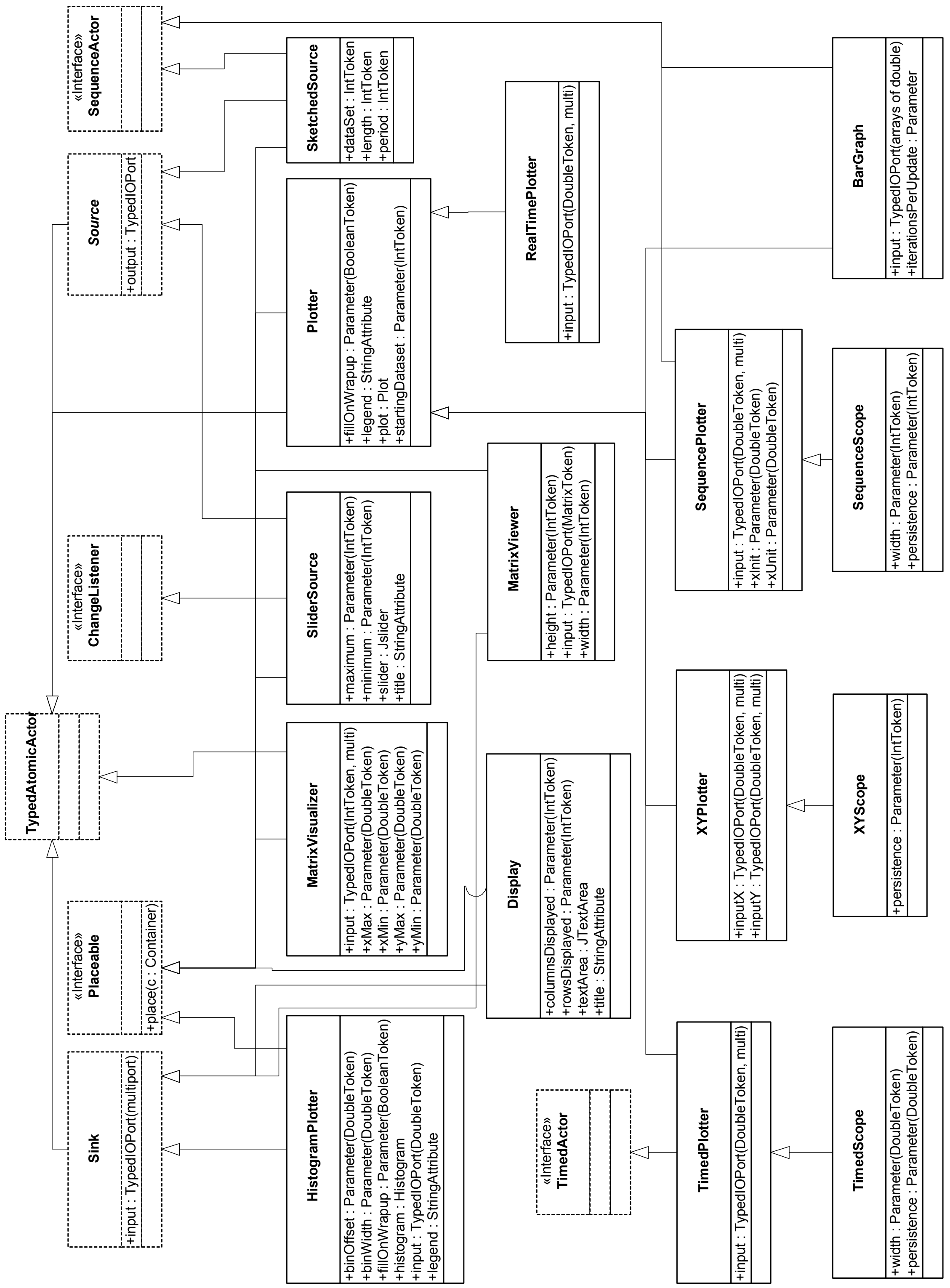
«Interface»
SequenceActor
Source
+output : TypedIOPort
«Interface»
ChangeListener
TypedAtomicActor
«Interface»
Placeable
+place(c : Container)
Sink
+input : TypedIOPort(multiport)
SketchedSource
+dataSet : IntToken
+length : IntToken
+period : IntToken
Plotter
+fillOnWrapup : Parameter(BooleanToken)
+legend : StringAttribute
+plot : Plot
+startingDataset : Parameter(IntToken)
SliderSource
+maximum : Parameter(IntToken)
+minimum : Parameter(IntToken)
+slider : Jslider
+title : StringAttribute
MatrixVisualizer
+input : TypedIOPort(IntToken, multi)
+xMax : Parameter(DoubleToken)
+xMin : Parameter(DoubleToken)
+yMax : Parameter(DoubleToken)
+yMin : Parameter(DoubleToken)
HistogramPlotter
+binOffset : Parameter(DoubleToken)
+binWidth : Parameter(DoubleToken)
+fillOnWrapup : Parameter(BooleanToken)
+histogram : Histogram
+input : TypedIOPort(DoubleToken)
+legend : StringAttribute
RealTimePlotter
+input : TypedIOPort(DoubleToken, multi)
MatrixViewer
+height : Parameter(IntToken)
+input : TypedIOPort(MatrixToken)
+width : Parameter(IntToken)
Display
+columnsDisplayed : Parameter(IntToken)
+rowsDisplayed : Parameter(IntToken)
+textArea : JTextArea
+title : StringAttribute
«Interface»
TimedActor
BarGraph
+input : TypedIOPort(arrays of double)
+iterationsPerUpdate : Parameter
SequencePlotter
+input : TypedIOPort(DoubleToken, multi)
+xInit : Parameter(DoubleToken)
+xUnit : Parameter(DoubleToken)
SequenceScope
+width : Parameter(IntToken)
+persistence : Parameter(IntToken)
XYPlotter
+inputX : TypedIOPort(DoubleToken, multi)
+inputY : TypedIOPort(DoubleToken, multi)
XYScope
+persistence : Parameter(IntToken)
TimedPlotter
+input : TypedIOPort(DoubleToken, multi)
TimedScope
+width : Parameter(DoubleToken)
+persistence : Parameter(DoubleToken)

FIGURE 4.3. Organization of actors in the ptolemy.actor.lib.gui package.

value is true (the default), then at the conclusion of the execution of the model, the axes of the plot will be adjusted to just fit the observed data.

All of the sink actors implement the Placeable interface. Actors that implement this interface have graphical widgets that a user of the actor may wish to place on the screen. Vergil constructs a display panel by placing such actors. More customized placement can be achieved by calling the place() method of the Placeable interface in custom Java code.

In Vergil, the sinks library is divided into *generic sinks*, *timed sinks*, and *sequence sinks*. The first group includes sinks that are agnostic about whether their inputs are interpreted as timed events or as sequence inputs. The other two groups contain actors for which the input is either timed or is logically a sequence.

Generic Sinks

Discard (extends *Sink*): Consume and discard input tokens.

Display (extends *Sink*): Display input tokens in a text area on the screen.

MonitorValue (extends *Sink*): Display input tokens in the icon of the actor in the block diagram. The *value* parameter specifies what to display before any inputs are provided.

Recorder (extends *Sink*): Record all input tokens for later querying (by Java code). This actor is useful for Java code that executes models and then wishes to query for results.

SetVariable (extends *TypedAtomicActor*): Set the value of a variable contained by the container. The change to the value of the variable takes hold at the end of the current iteration. This helps ensure that users of value of the variable will see changes to the value deterministically (independent of the schedule of execution of the actors).

XYPlotter (extends *Plotter*): Display a plot of the data on each *inputY* channel vs. the data on the corresponding *inputX* channel.

XYScope (extends *XYPlotter*): Display a plot of the data on each *inputY* channel vs. the data on the corresponding *inputX* channel with finite persistence.

Timed Sinks

TimedPlotter (extends *Plotter*): Plot inputs as a function of time.

TimedScope (extends *TimedPlotter*): Plot inputs as a function of time in an oscilloscope style.

Sequence Sinks

ArrayPlotter (extends *Plotter*): Plot a sequence of arrays of doubles.

BarGraph (extends *ArrayPlotter*): Plot bar graphs, given arrays of doubles as inputs.

HistogramPlotter (extends *PlotterBase*): Display a histogram of the data on each input channel.

SequencePlotter (extends *Plotter*): Plot the input tokens vs. their index number.

SequenceScope (extends *SequencePlotter*): Plot sequences that are potentially infinitely long in an oscilloscope style.

4.3.3 Array

The array library supports manipulations of arrays, which are ordered collections of tokens of arbitrary type.

ArrayAppend (extends *Transformer*): Append arrays on the input channels to produce a single output array.

ArrayAverage (extends *Transformer*): Output the average of the input array.

ArrayElement (extends *Transformer*): Extract an element from an array and produce it on the output.

ArrayExtract (extends *Transformer*): Extract a subarray from an array and produce it on the output.

ArrayLength (extends *Transformer*): Output the length of the input array.

ArrayLevelCrossing (extends *TypedAtomicActor*): Find and output the index of the first item in an input array to cross a specified threshold.

ArrayMaximum (extends *Transformer*): Extract the maximum element from an array.

ArrayMinimum (extends *Transformer*): Extract the minimum element from an array.

ArrayPeakSearch (extends *TypedAtomicActor*): Output the indices and values of peaks in an input array.

ArraySort (extends *Transformer*): Sort the elements of an input array.

ArrayToElements (extends *Transformer*): Send out each element of an input array to the corresponding channel of the output port.

ArrayToSequence (extends *SDFTransformer*): Extract all elements from an *input* array and produce them sequentially on the output port.

ElementsToArray (extends *Transformer*): Read exactly one token from each channel of the input port, assemble the tokens into an array and send it to the output port.

SequenceToArray (extends *SDFTransformer*): Collect a sequence of inputs into an array and produce the array on the output port.

4.3.4 Conversions

Ptolemy II has a sophisticated type system that allows actors to be polymorphic (to operate on multiple data types). Typically, actors express type constraints between their ports and their parameters. When actors are connected, these type constraints are resolved to determine the type of each port. Conversions between types are automatic if they result in no loss of data. However, sometimes, a model builder may wish to force a particular conversion. The actors in the conversions library support this.

BooleanToAnything (extends *Converter*): Convert a Boolean input token to any data type.

BitsToInt (extends *SDFConverter*): Convert 32 successive binary inputs into a two's complement integer.

CartesianToComplex (extends *TypedAtomicActor*): Convert two tokens representing the real and imaginary of a complex number into their complex representation.

CartesianToPolar (extends *TypedAtomicActor*): Convert a Cartesian pair (a token on the *x* input and a token on the *y* input) to two tokens representing its polar form (which are output on *angle* and *magnitude*).

ComplexToCartesian (extends *TypedAtomicActor*): Convert a token representing a complex number into its Cartesian components (which are output on *real* and *imag*).

ComplexToPolar (extends *TypedAtomicActor*): Convert a token representing a complex number into two tokens representing its polar form (which are output on *angle* and *magnitude*).

DoubleToFix (extends *Converter*): Convert a double into a fix point number with a specific precision, using a specific quantization strategy.

ExpressionToToken (extends *Converter*): Read a string expression from the input port and outputs the token resulting from the evaluation.

FixToDouble (extends *Converter*): Convert a fix point into a double, by first setting the precision of the fix point to the supplied precision, using a specific quantization strategy.

FixToFix (extends *Converter*): Convert a fix point into another fix point with possibly a different precision, using a specific quantizer and overflow strategy.

IntToBits (extends *SDFConverter*): Convert an input integer into 32 successive binary outputs.

InUnitsOf (extends *Transformer*): Convert input tokens to specified units by dividing the input by the value of the *units* parameter. This actor is designed to be used with a *unit system*, which must be included in the model (note that some Ptolemy II applications do not include unit systems).

LongToDouble (extends *Converter*): Convert an input of type *long* to an output of type *double*.

PolarToCartesian (extends *TypedAtomicActor*): Converts two tokens representing a polar coordinate (a token on *angle* and a token on *magnitude*) to two tokens representing their Cartesian form (which are output on *x* and *y*).

PolarToComplex (extends *TypedAtomicActor*): Converts two tokens representing polar coordinates (a token on *angle* and a token on *magnitude*) to a token representing their complex form.

Round (extends *TypedAtomicActor*): Produce an output token with a value that is a rounded version of the input. The rounding method is specified by the *function* attribute, where valid functions are *ceil*, *floor*, *round*, and *truncate*.

StringToUnsignedByteArray (extends *Converter*): Convert an input of type *string* to an array of type *unsignedByte*.

TokenToExpression (extends *Converter*): Read a string expression from the input port and output the token resulting from the evaluation.

UnsignedByteArrayToString (extends *Converter*): Convert an input that is an array of bytes into a string.

4.3.5 Flow Control

The flow control actors route tokens or otherwise affect the flow of control.

Aggregators

BusAssembler (extends *TypedAtomicActor*): Assemble input port channels into output bus.

BusDisassembler (extends *TypedAtomicActor*): Split input bus channels onto output port channels.

Commutator (extends *Transformer*): Interleave the data on the input channels into a single sequence on the output.

Distributor (extends *Transformer*): Distribute the data on the input sequence into multiple sequences on the output channels.

Multiplexor (extends *Transformer*): Produce as output the token on the channel of *input* specified by the *select* input. Exactly one token is consumed from each channel of *input* in each firing.

RecordAssembler (extends *TypedAtomicActor*): Produce an output token that results from combining a token from each of the input ports (which must be added by the user). To add input ports to the actor in Vergil, right click on its icon and select "Configure Ports," and then select "Add." The name of each field in the record is the name of the port that supplies the field.

RecordDisassembler (extends *TypedAtomicActor*): Produce output tokens on the output ports (which must be added by the user) that result from separating the record on the input port. To add output ports to the actor in Vergil, right click on its icon and select "Configure Ports," and then select "Add." The name of each field extracted from the record is the name of the output port to which the value of the field is sent.

RecordUpdater (extends *TypedAtomicActor*): Produce an output token that results from the union of the record read from the *input* port and the values supplied by the other input ports. The user must create the other input ports. Input ports with the same name as a field in the original input record are used to update the corresponding field in the output token.

Select (extends *Transformer*): Produce as output the token on the channel of *input* specified by the *control* input. Tokens on channels that are not selected are not consumed.

Switch (extends *Transformer*): Produce the token read from the *input* port on the channel of *output* specified by the *control* input.

Synchronizer (extends *Transformer*): Wait until at least one token exists on each channel of *input*, then consume exactly one token from each input channel and output each token on its corresponding output channel.

VectorAssembler (extends *Transformer*): On each firing, read exactly one token from each channel of the *input* port and assemble the tokens into a double matrix with one column.

VectorDisassembler (extends *Transformer*): On each firing, read one column vector (i.e. a double matrix token with one column) from the *input* port and send out individual doubles to each channel of the *output* port.

Boolean Flow Control

BooleanMultiplexor (extends *TypedAtomicActor*): Produce as output the token from either *trueInput* or *falseInput* as specified by the *select* input. Exactly one token from each input port is consumed.

BooleanSelect (extends *TypedAtomicActor*): Produce as output the token from either *trueInput* or *falseInput* as specified by the *control* input. Tokens from the port that is not selected are not consumed.

BooleanSwitch (extends *TypedAtomicActor*): Produce the token read from the *input* port on either the *trueOutput* or the *falseOutput* port, as specified by the *control* input port.

CountTrues (extends *SDFTransformer*): Read the specified number of input booleans and output the number that are true.

Sequence Control

Chop (extends *SDFTransformer*): Chop an input sequence and construct from it a new output sequence. This actor can be used, for example, for zero-padding, overlapping windows, delay lines, etc.

Repeat (extends *SDFTransformer*): Repeat each input sample (a block of tokens) a specified number of times.

SampleDelay (extends *SDFTransformer*): Produce a set of initial tokens during the initialize() method, and subsequently pass input tokens to the output. Used to break dependency cycles in directed loops of SDF models.

Sequencer (extends *Transformer*): Put tokens in order according to their numbers in a sequence.

Execution Control

Stop (extends *Sink*): Stop execution of a model when it receives a *true* token on any input channel.

ThrowException (extends *Sink*): Throw an *IllegalActionException* when it receives a *true* token on any input channel.

ThrowModelError (extends *Sink*): Throw a model error when it receives a *true* token on any input channel. A model error is an exception that is passed up the containment hierarchy rather than being immediately thrown as an exception.

4.3.6 Higher Order Actors

Most actors in Ptolemy II have parameters (or inputs) that allow users to control the computation performed by the actors. Such parameters usually have “simple” values, such as integers, records, and matrices. A higher order actor may have a parameter that is a reference to another model, or an input that receives specifications from which submodels are created.

MobileFunction (extends *TypedAtomicActor*): Apply a function to the input and output the result. The function is defined by the most recent function token received by the actor from its *function* input. Before the first function is received, the identity function is applied. Currently, only functions with one argument are supported.

MobileModel (extends *TypedCompositeActor*): A MobileModel actor delegates the computation to a submodel that can be changed during execution. The submodel is changed when a string token is received from the *modelString* input of the actor. The string token contains the MoML (see the MoML chapter for details) description of the submodel. The *input* and *output* of the actor is connected to the corresponding port of the submodel. Currently, it only accepts models with one input and one output, and requires that the model name its input port as "input" and output port as "output."

ModalModel: This is a typed composite actor designed to be a modal model. Inside the modal model is a finite-state machine controller, and inside each state of the FSM is a refinement model. To use this actor, just drag it into a model, and look inside to start constructing the controller. You may add ports to get inputs and outputs, and add states to the controller. You may add one or more refinements to a state. Each refinement is required to have its own director. See the Modal Model section in the FSM Domain chapter for more details.

ModelReference (extends *TypedAtomicActor*): This is an atomic actor that can execute a model specified by a file or URL. This can be used to define an actor whose firing behavior is given by a complete execution of another model. An instance of this actor can have ports added to it. If it has input ports, then on each firing, before executing the referenced model, the actor will read an input token from each input port, if there is one, and use the token to set the value of a top-level parameter in the referenced model that has the same name as the port, if there is one. Input ports should not be multiports, and if they are, then all but the first channel will be ignored. If this actor has output ports and the referenced model is executed, then upon completion of that execution, this actor looks for top-level parameters in the referenced model whose names match those of the output ports. If there are such parameters, then the final value of those parameters is sent to the output ports. Normally, when you create output ports for this actor, you will have to manually set the type. There is no type inference from the parameters of the referenced model.

MultiInstanceComposite (extends *TypedCompositeActor*): A MultiInstanceComposite actor may be used to instantiate *nInstances* identical processing blocks within a model. This actor (the "master") creates *nInstances* − 1 additional instances (clones) of itself during the preinitialize phase of model execution and destroys these additional instances during model wrapup. MultiInstanceComposite must be opaque (have a local director). Each instance may refer to its *instance* parameter which is set automatically between 0 and nInstances-1 by the master if it needs to know its instance number.

RunCompositeActor (extends *LifeCycleManager*): This is a composite actor that can execute the contained model completely, as if it were a top-level model, on each firing. This can be used to define an actor whose firing behavior is given by a complete execution of a submodel. An instance of this actor can have ports added to it. On each firing, if there is a token at an input port, and the actor has a parameter with the same name as the port, then the token is used to set the value of the parameter. The simplest way to ensure that there is a matching parameter is to use a PortParameter for inputs. However, this actor will also work with ordinary ports. Input ports should not be multiports, and if they are, then all but the first channel will be ignored. Upon completion of executing the contained model, if this actor has parameters whose names match those of the output ports, then the final value of those parameters is sent to the output ports.

VisualModelReference (extends *ModelReference*): This actor extends the base class with the capability to open the referenced model in a Vergil window.

4.3.7 I/O

The IO library (see figure 4.2) consists of actors that read and write to the file system or network. Note that the "comm" library under "more libraries" includes a Windows only *SerialComm* actor that communicates with serial ports.

ArrowKeySensor (extends *TypedAtomicActor*): Pop up a frame that senses arrow keystrokes and produces outputs accordingly.

DatagramReader (extends *TypedAtomicActor*): Read datagram packets from the network socket specified by localSocketNumber and produce them on the output.

DatagramWriter (extends *TypedAtomicActor*): Send input data received on *data* port as a UDP datagram packet to the network address specified by *remoteAddress* and *remoteSocketNumber*.

DirectoryListing (extends *Source*): Output an array that lists the contents of a directory.

ExpressionReader (extends *LineReader*): Read a file or URL, one line at a time, evaluate each line as an expression, and output the token resulting from the evaluation.

ExpressionWriter (extends *LineWriter*): Read input tokens and write them, one line at a time, to a specified file.

FileReader (extends *Source*): Read a file or URL and output the entire content as a single string.

LineReader (extends *Source*): Read a file or URL, one line at a time, and output each line as a string token.

LineWriter (extends *Sink*): Read input string-valued tokens and write them, one line at a time, to a specified file.

4.3.8 Logic

The logic actors perform logical operations on inputs.

Comparator (extends *TypedAtomicActor*): Produce an output token with a value that is a comparison of the input. The comparison is specified by the *comparison* attribute, where valid comparisons are >, >=, <, <=, and ==.

Equals (extends *Transformer*): Consume at most one token from each channel of *input*, and produce an output token with value true if these tokens are equal in value, and false otherwise.

IsPresent (extends *Transformer*): Consume at most one token from each channel of *input*, and output a boolean on the corresponding output channel (if there is one). The value of the boolean is true if the input is present and false otherwise.

LogicalNot (extends *Transformer*): Produce an output token which is the logical negation of the input token.

LogicFunction (extends *Transformer*): Produce an output token with a value that is a logical function of the tokens on the channels of *input*. The function is specified by the *function* attribute, where valid functions are *and*, *or*, *xor*, *nand*, *nor*, and *xnor*.

4.3.9 Math

The Math library (see figure 4.2) consists mostly of transformer actors, each of which calculates some mathematical function.

AbsoluteValue (extends *Transformer*): Produce an output on each firing with a value that is equal to the absolute value of the input.

AddSubtract (extends *TypedAtomicActor*): Add tokens on the *plus* input channels and subtract tokens on the *minus* input channels.

Accumulator (extends *Transformer*): Output the initial value plus the sum of all the inputs since the last time a true token was received at the *reset* port.

Average (extends *Transformer*): Output the average of the inputs since the last time a true token was received at the *reset* port. The *reset* input may be left disconnected in most domains.

Counter (extends *TypedAtomicActor*): An up-down counter of received tokens.

Differential (extends *Transformer*): Output the difference between successive inputs.

DotProduct (extends *TypedAtomicActor*): Output the dot product of two input arrays.

Expression (extends *TypedAtomicActor*): On each firing, evaluate the *expression* parameter, whose value is set by an expression that may include references to any input ports that have been added to the actor. The expression language is described in the Expressions chapter, with the addition that the expression can refer to the values of inputs, and to the current time by the variable named “time,” and to the current iteration count by the variable named “iteration.” To add input ports to the actor in Vergil, right click on its icon and select “Configure Ports,” and then select “Add.”

Limiter (extends *Transformer*): Produce an output token on each firing with a value that is equal to the input if the input lies between *top* and *bottom*. Otherwise, if the input is greater than *top*, output *top*. If the input is less than *bottom*, output *bottom*.

LookupTable (extends *Transformer*): Output the value in the array of tokens specified by the *table* parameter at the index specified by the *input* port.

MathFunction (extends *TypedAtomicActor*): Produce an output token with a value that is a function of the input(s). The function is specified by the *function* attribute, where valid functions are *exp*, *log*, *modulo*, *sign*, *square*, and *sqrt*.

Maximum (extends *TypedAtomicActor*): Broadcast an output token on each firing on *maximumValue* with a value that is the maximum of the values on the input channels. The index of this maximum is broadcast on *channelNumber*.

Minimum (extends *TypedAtomicActor*): Broadcast an output token on each firing on *minimumValue* with a value that is the minimum of the values on the input channels. The index of this minimum is broadcast on *channelNumber*.

MultiplyDivide (extends *TypedAtomicActor*): Multiply tokens on the *multiply* input channels, and divide by tokens on the *divide* input channels.

Quantizer (extends *Transformer*): Produce an output token with the value in *levels* that is closest to the input value.

Remainder (extends *Transformer*): Produce an output token with the value that is the remainder after dividing the token on the *input* port by the *divisor*.

Scale (extends *Transformer*): Produce an output that is the product of the *input* and the *factor*.

TrigFunction (extends *Transformer*): Produce an output token with a value that is a function of the input. The function is specified by the *function* attribute, where valid functions are *acos*, *asin*, *atan*, *cos*, *sin*, and *tan*.

4.3.10 Matrix

The matrix library supports matrix manipulations. Currently this library is very small; if you need matrix operations that are not in this library, then very likely they are available in the expression language (see the Expression chapter). You can access these using the *Expression* actor.

MatrixToSequence (extends *SDFTransformer*): Unbundle a matrix into a sequence of output tokens. On each firing, this actor writes the elements of the matrix to the output as a sequence of output tokens.

MatrixViewer (extends *Sink*): Display the contents of a matrix input.

SequenceToMatrix (extends *SDFTransformer*): Bundle a specified number of input tokens into a matrix. On each firing, this actor reads *rows* times *columns* input tokens and writes one output matrix token with the specified number of rows and columns.

4.3.11 Random

The random library (see figure 4.2) consists of actors that generate random data. All actors in this library have a *seed* parameter. A seed of zero is interpreted to mean that no seed is specified. In such cases, a seed based on the current machine time and the actor instance is used to make it unlikely that two identical sequences are produced.

Bernoulli (extends *RandomSource*): Produce a random sequence of booleans (a source of coin flips).

DiscreteRandomSource (extends *RandomSource*): Produce tokens with the given probability mass function.

Gaussian (extends *RandomSource*): Produce a random sequence with a Gaussian distribution.

Rician (extends *RandomSource*): Produce a random sequence with a Rician or Rayleigh distribution.

Uniform (extends *RandomSource*): Produce a random sequence with a uniform distribution.

4.3.12 Real Time

The behavior of the real time actors is affected by the amount of elapsed real time.

RealTimePlotter (extends *Plotter*): Plot input data as a function of elapsed real time.

Sleep (extends *Transformer*): Produce as output the tokens received on input after an amount of real time specified by the *sleepTime* parameter.

VariableSleep (extends *Transformer*): Produce as output the tokens received on input after an amount of real time specified by the *sleepTime* input. NOTE: This will likely be replaced by a version of *Sleep* with a PortParameter.

WallClockTime (extends *Source*): Output the elapsed real time in seconds.

4.3.13 Signal Processing

The signal processing library is divided into sublibraries.

Audio

The audio library provides actors that can read and write audio files, can capture data from an audio input such as a CD or microphone, and can play audio data through the speakers of the computers. It uses the javasound library, which is part of the Sun Microsystems' Java 2 Standard Edition (J2SE) version 1.3.0 and higher. The AudioCapture and AudioPlayer actors are unusual in that they have coupled parameter values. Changing the parameters of one results in the parameters of the other being changed. Also, as of this writing, they have the restriction that only one of each may be used in a model at a time, and that if there are two models that use them, then those two models may not be executed simultaneously.

AudioCapture (extends *Source*): Capture audio from the audio input port of the computer, or from its microphone, and produce the samples at the output.

AudioReader (extends *Source*): Read audio from a URL, and produce the samples at the output.

AudioPlayer (extends *Sink*): Play audio samples on the audio output port of the computer, or from its speakers.

AudioWriter (extends *Sink*): Write audio data to a file.

Communications

The communications library collects actors that support modeling and design of digital communication systems.

ConvolutionalCoder (extends *Transformer*): Encode an input sequence of bits using a convolutional code.

DeScrambler (extends *Transformer*): Descramble the input bit sequence using a feedback shift register.

HadamardCode (extends *Source*): Produce a Hadamard codeword by selecting a row from a Hadamard matrix.

HammingCoder (extends *Transformer*): Encode an input sequence of bits using Hamming code.

HammingDecoder (extends *Transformer*): Decode an input sequence of bits using Hamming code.

LineCoder (extends *SDFTransformer*): Read a sequence of booleans (of length *wordLength*) and interpret them as a binary index into the *table*, from which a token is extracted and sent to the output.

LMSAdaptive (extends *FIR*): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.

RaisedCosine (extends *FIR*): An FIR filter with a raised cosine frequency response. This is typically used in a communication systems as a pulse shaper or a matched filter.

Scrambler (extends *Transformer*): Scramble the input bit sequence using a feedback shift register.

Slicer (extends *Transformer*): A decoder of the LineCoder.

TrellisDecoder (extends *ViterbiDecoder*): Decode convolutional code with non-antipodal constellation.

ViterbiDecoder (extends *Transformer*): Decode inputs using (hard or soft) Viterbi decoding.

Filtering

DelayLine (extends *SDFTransformer*): In each firing, output the *n* most recent input tokens collected into an array, where *n* is the length of *initialValues*. In the beginning, before there are *n* most recent tokens, use the tokens from *initialValues*.

DownSample (extends *SDFTransformer*): Read *factor* inputs and produce only one of them on the output.

FIR (extends *SDFTransformer*): Produce an output token with a value that is the input filtered by an FIR filter with coefficients given by *taps*.

GradientAdaptiveLattice (extends *Lattice*): A lattice filter that adapts the reflection coefficients to minimize the power of the output sequence.

IIR (extends *Transformer*): Produce an output token with a value that is the input filtered by an IIR filter using a direct form II implementation.

Lattice (extends *Transformer*): Produce an output token with a value that is the input filtered by an FIR lattice filter with coefficients given by *reflectionCoefficients*.

LinearDifferenceEquationSystem (extends *Transformer*): Linear system given by an [A, b, c, d] state-space model.

LMSAdaptive (extends *FIR*): Filter the input with an adaptive filter, and update the coefficients of the filter using the input error signal according to the LMS (least mean-square) algorithm.

RecursiveLattice (extends *Transformer*): Produce an output token with a value that is the input filtered by a recursive lattice filter with coefficients given by reflectionCoefficients.

UpSample (extends *SDFTransformer*): Read one input token and produce *factor* outputs, with all but one of the outputs being a zero of the same type as the input.

VariableFIR (extends *FIR*): Filter the input sequence with an FIR filter with coefficients given on the *newTaps* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newTaps*.

VariableLattice (extends *Lattice*): Filter the input sequence with an FIR lattice filter with coefficients given on the *newCoefficients* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newCoefficients*.

VariableRecursiveLattice (extends *Lattice*): Filter the input sequence with a recursive lattice filter with coefficients given on the *newCoefficients* input port. The *blockSize* parameter specifies the number of successive inputs that are processed for each set of taps provided on *newCoefficients*.

Spectrum

DB (extends *Transformer*): Produce a token that is the value in decibels ($k*\log_{10}(z)$) of the token received, where k is 10 if *inputIsPower* is true, and 20 otherwise. The output is never less than *min* (it is clipped if necessary).

FFT (extends *SDFTransformer*): A fast Fourier transform of size 2^{order}.

IFFT (extends *SDFTransformer*): An inverse fast Fourier transform of size 2^{order}.

LevinsonDurbin (extends *SDFTransformer*): Calculate the linear predictor coefficients (for both an FIR and Lattice filter) for the specified autocorrelation input.

MaximumEntropySpectrum (composite actor): A fancy spectrum estimator that uses the Levinson-Durbin algorithm to calculate linear predictor coefficients, and then uses those as a parametric model for the random process.

Periodogram (composite actor): A spectrum estimator calculates a periodogram.

PhaseUnwrap (extends *Transformer*): A simple phase unwrapper.

SmoothedSpectrum (composite actor): A spectrum estimator called the Blackman-Tukey algorithm, which estimates an autocorrelation function by averaging products of the input samples, and then calculates the FFT of that estimate.

Spectrum (composite actor): A simple spectrum estimator that calculates the FFT of the input. For a random process, this is called the periodogram spectral estimate.

Statistical

A small number of statistical analysis actors are provided.

Autocorrelation (extends *SDFTransformer*): Estimate the autocorrelation by averaging products of the input samples.

ComputeHistogram (extends *TypedAtomicActor*): Compute a histogram of input data.

PowerEstimate (extends *Transformer*): Estimate the power of the input signal.

4.3.14 String

The String library consists of actors that operate on strings.

StringCompare (extends *TypedAtomicActor*): Compute a specified string comparison function on the two string inputs. The function is specified by the *function* attribute, where valid functions are *equals*, *startsWith*, *endsWith*, and *contains*.

StringFunction (extends *Transformer*): Apply a specified function on the input string and send the result to the output. The function is specified by the *function* attribute, where valid functions are *toLowerCase*, *toUpperCase*, and *trim*.

StringIndexOf (extends *TypedAtomicActor*): Output the index of a string (*searchFor*) contained in another string (*inText*).

StringLength (extends *Transformer*): Output the length of an input string.

StringMatches (extends *TypedAtomicActor*): Output true if *matchString* matches *pattern*, false otherwise.

StringReplace (extends *TypedAtomicActor*): Replace a substring of *stringToEdit* that matches *pattern* by *replacement*. If *replaceAll* is true, then all matching substrings are replaced.

StringSubstring (extends *Transformer*): Output a substring of the input string, from the *start* index to *stop*.

4.3.15 Domain Specific

Several sublibraries contain actors that are primarily useful only with corresponding directors.

Continuous Time

The continuous-time library contains a set of actors designed specifically for use in the CT domain.

ContinuousClock: Generate a piecewise-constant signal with instantaneous transitions between levels.

TriggeredContinuousClock: Generate a piecewise-constant signal with instantaneous transitions between levels, where two input ports are provided to start and stop the clock.

ContinuousSinewave: Generate a continuous-time sinusoidal signal.

CTCompositeActor: Composite actor to use when a continuous-time model is created within a continuous-time model.

Continuous Time: Dynamics

The actors in this sublibrary have continuous-time dynamics (i.e., they involve integrators, and hence must coordinate with the differential equation solver).

Integrator: Integrate the input signal over time to produce the output signal. That is, the input is the derivative of the output with respect to time. This actor can be used to close feedback loops in CT to define interesting differential equation systems.

LaplaceTransferFunction: Filter the input with the specified rational Laplace transform transfer function. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

LinearStateSpace: Filter the input with a linear system. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

DifferentialSystem: Filter the input with the specified system, which can nonlinear, and is specified using the expression language. Note that this actor constructs a submodel, so it might be interesting to look inside the actor after it is initialized.

RateLimiter: Limit the first derivative of the input signal, so that the output changes no faster than the specified limit.

Continuous Time: To Discrete

The actors in this sublibrary produce discrete event signals, which are signals that only have values at discrete points in time.

EventSource: Output a set of events at discrete set of time points.

LevelCrossingDetector: A event detector that converts continuous signals to discrete events when the continuous signal crosses a level threshold.

PeriodicSampler: Sample the input signal with the specified rate, producing discrete output events.

TriggeredSampler: Sample the input signal at times where the trigger input has a discrete input events.

ThresholdMonitor: Output *true* if the input value is in the interval [*a*, *b*], which is centered at *thresholdCenter* and has width *thresholdWidth*. This actor controls the integration step size so that the input does not cross the threshold without producing at least one *true* output.

ZeroCrossingDetector: When the *trigger* is zero (within the specified *errorTolerance*), then output the value from the *input* port as a discrete event. This actor controls the integration step size to accurately resolve the time at which the zero crossing occurs.

Continuous Time: To Continuous

The actors in this sublibrary convert discrete event signals into continuous-time signals.

FirstOrderHold: Convert discrete events at the input to a continuous-time signal at the output by projecting the value with the derivative.

ZeroOrderHold: Convert discrete events at the input to a continuous-time signal at the output by holding the value of the discrete event until the next discrete event arrives.

4.3.16 Discrete Event

A library of actors is provided to particularly support discrete-event models. In discrete-event models, signals consist of events placed in time, where time is a double. Events are processed in chronological order.

EventButton: Output a token in response to the click of a button.

EventFilter: An actor that filters a stream of boolean tokens. Every true input token that it receives is reproduced on the output port. False tokens are discarded. This is usually used to properly trigger other discrete event actors (such as inhibit and select) based on boolean values.

Inhibit: Output a received input token, unless the inhibit port receives a token.

Merge: Merge input events into a single signal.

PreemptableTask: Simulate a preemptable task.

Previous: On each iteration, this actor produces the token received on the previous iteration. On the first iteration, it produces the token given by the *initialValue* parameter, if such a value has been set.

Queue: This actor implements an event queue. When a token is received on the input port, it is stored in the queue. When the trigger port receives a token, the oldest element in the queue is output. If there is no element in the queue when a token is received on the trigger port, then no output is produced.

QueueWithNextOut: This actor is like the *Queue* actor above. An additional output port, *nextOut*, has been added which allows the model to know what's next to come out. This new output produces a token whenever the queue has been empty and a new token is queued. It also produces an output whenever a token is taken from the queue and at least one token remains. Otherwise, no output token is produced at *nextOut*. The token produced is the oldest token remaining in the queue.

Sampler: On each *trigger* input, produce at the output the most recently seen input.

Server: Delay input events until they have been “served” for the specified amount of time.

SingleEvent: Produce a single event with the specified time and value.

TimedDelay: Delay input events by the specified amount.

TimeGap: Produce at the output the amount of time between input events.

Timer: Given an input time value, produce *value* on the output that amount of time in the future.

VariableDelay: Delay input events by the specified amount.

WaitingTime: Measure the amount of time that one event (arriving on *waiter*) has to wait for an event to arrive on *waitee*. There is an output event for every event that arrives on *waiter*, where the value of that output is the time spent waiting, and the time of the output is time of the arriving *waitee* event.

4.4 Data Polymorphism

A data polymorphic actor is one that can operate on any of a number of input data types. For example, AddSubtract can accept any numeric type of input.

Figure 4.4 shows the methods defined in the base class Token. Any data exchanged between actors in Ptolemy II is wrapped in an instance of Token (or more precisely, in an instance of a class derived from Token). Notice that add() and subtract() are methods of this base class. This makes it easy to implement a data polymorphic adder.

It is instructive to examine the code in an actor that performs data polymorphic operations. The fire() method of the AddSubtract actor is shown in figure 4.5. In this code, we first iterate through the channels of *plus* input. The first token read (by the get() method) is assigned to sum. Subsequently, the polymorphic add() method of that token is used to add additional tokens. The second iteration, over the channels at the *minus* input port, is slightly trickier. If no tokens were read from the *plus* input, then the variable sum is initialized by calling the polymorphic zero() method of the first token read at the *minus* port. The zero() method returns whatever a zero value is for the token in question.

Not all classes derived from Token override all its methods. For example, StringToken overrides add() but not subtract(). Adding strings means simply concatenating them, but it is hard to assign a rea-

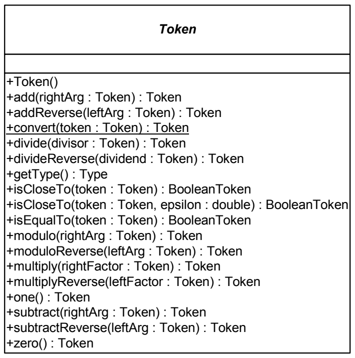
Token
+Token()
+add(rightArg : Token) : Token
+addReverse(leftArg : Token) : Token
+convert(token : Token) : Token
+divide(divisor : Token) : Token
+divideReverse(dividend : Token) : Token
+getType() : Type
+isCloseTo(token : Token) : BooleanToken
+isCloseTo(token : Token, epsilon : double) : BooleanToken
+isEqualTo(token : Token) : BooleanToken
+modulo(rightArg : Token) : Token
+moduloReverse(leftArg : Token) : Token
+multiply(rightFactor : Token) : Token
+multiplyReverse(leftFactor : Token) : Token
+one() : Token
+subtract(rightArg : Token) : Token
+subtractReverse(leftArg : Token) : Token
+zero() : Token

FIGURE 4.4. The Token class defines a polymorphic interface that includes basic arithmetic operations.

```
public void fire() throws IllegalActionException {
    Token sum = null;
    for (int i = 0; i < plus.getWidth(); i++) {
        if (plus.hasToken(i)) {
            if (sum == null) {
                sum = plus.get(i);
            } else {
                sum = sum.add(plus.get(i));
            }
         }
    }
    for (int i = 0; i < minus.getWidth(); i++) {
        if (minus.hasToken(i)) {
            Token in = minus.get(i);
            if (sum == null) {
                sum = in.zero();
            }
            sum = sum.subtract(in);
        }
    }
    if (sum != null) {
        output.send(0, sum);
    }
}
```

FIGURE 4.5. The fire() method of the AddSubtract shows the use of polymorphic add() and subtract() methods in the Token class (see figure 4.4).

sonable meaning to subtraction. Thus, if AddSubtract is used on strings, then the *minus* port must not ever receive tokens. It may be simply left disconnected, in which case minus.getWidth() returns zero. If the subtract() method of a StringToken is called, then a runtime exception will be thrown.

4.5 Domain Polymorphism

Most actors access their ports as shown in figure 4.5, using the hasToken(), get(), and send() methods. Those methods are polymorphic, in that their exact behavior depends on the domain. For example, get() in the CSP domain causes a rendezvous to occur, which means that the calling thread is suspended until another thread sends data to the same port (using, for example, the send() method on one of its ports). Correspondingly, a call to send() causes the calling thread to suspend until some other thread calls a corresponding get(). In the PN domain, by contrast, send() returns immediately (if there is room in the channel buffers), and only get() causes the calling thread to suspend.

Each domain has slightly different behavior associated with hasToken(), get(), send() and other methods of ports. The actor, however, does not really care. The fire() method shown in figure 4.5 will work for any reasonable implementation of these methods. Thus, the AddSubtract actor is domain polymorphic.

Domains also have different behavior with respect to when the fire() method is invoked. In process-oriented domains, such as CSP and PN, a thread is created for each actor, and an infinite loop is created to repeatedly invoke the fire() method. Moreover, in these domains, hasToken() always returns *true*, since you can call get() on a port and it will not return until there is data available. In the DE domain, the fire() method is invoked only when there are new inputs that happen to be the oldest ones in the model, and hasToken() returns *true* only if there is new data on the input port. The design of actors for multiple domains is covered in the Designing Actors chapter.