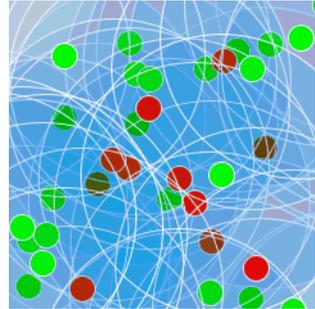


VISUALSENSE: VISUAL MODELING FOR WIRELESS AND SENSOR NETWORK SYSTEMS

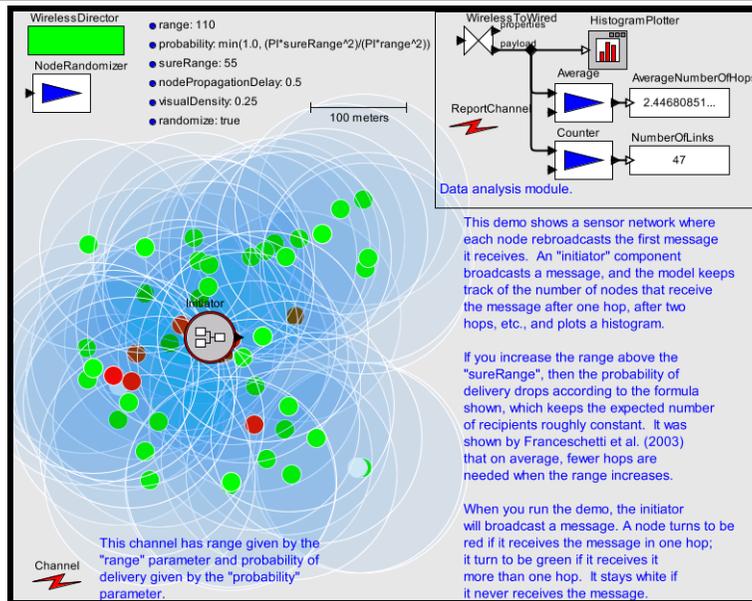
Authors¹: Philip Baldwin
 Sanjeev Kohli
 Edward A. Lee
 Xiaojun Liu
 Yang Zhao



Contributors: C. T. Ee
 Christopher Hylands Brooks
 N. V. Krishnan
 Stephen Neuendorffer
 Charlie Zhong
 Rachel Zhou

Version 4.0²
 UCB ERL Memorandum UCB/ERL M04/8
 April 23, 2004

This document describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from the State of California MICRO program, and the following companies: Daimler-Chrysler, Honeywell, Toyota and Wind River Systems.



1. With contributions from the entire Ptolemy II team.
2. The version number for VisualSense matches the version of Ptolemy II on which it is based.

Copyright (c) 2004 The Regents of the University of California.

All rights reserved.

Permission is hereby granted, without written agreement and without license or royalty fees, to use, copy, modify, and distribute the VisualSense software and its documentation for any purpose, provided that the above copyright notice and the following two paragraphs appear in all copies of the software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

- 1. Introduction 5**
 - 1.1. Installation and Quick Start 5
- 2. Modeling Wireless Networks 6**
 - 2.1. Running a Pre-Built Model 7
 - 2.2. Changing Parameters 8
 - 2.3. Structure of a Pre-Built Model 8
 - 2.3.1. *Visual Representations (Icons)* 9
 - 2.3.2. *Channels* 11
 - 2.3.3. *Composite Actors* 12
 - 2.4. Controlling the Execution 13
 - 2.5. Building a New Model 14
 - 2.6. Using the Plot Actors 22
- 3. Modeling Capabilities 23**
 - 3.1. Discrete-Event Simulation 24
 - 3.2. Channel Models 24
 - 3.3. Wireless Node Models 24
 - 3.4. Examples of Modeling Capabilities 25
 - 3.4.1. *Packet Structure* 25
 - 3.4.2. *Packet Losses* 25
 - 3.4.3. *Battery Power* 25
 - 3.4.4. *Power Loss* 25
 - 3.4.5. *Collisions* 25
 - 3.4.6. *Transmit Antenna Gain* 27
- 4. Software Architecture 32**
 - 4.1. Erasure Channel 34
 - 4.2. Limited Range Channels 35
 - 4.3. Transmit Properties 35
 - 4.4. Antenna Gains and Terrain Models 36
 - 4.5. Delay Channels 36
- 5. Framework Infrastructure 36**
 - 5.1. Hierarchy and Heterogeneity 37
 - 5.2. Type System 37
 - 5.3. Expressions 37
 - 5.3.1. *Expression Evaluator* 37
 - 5.3.2. *Simple Arithmetic Expressions* 38
 - Constants and Literals 38
 - Variables 40
 - Operators 40
 - Comments 42
 - 5.3.3. *Uses of Expressions* 42
 - Parameters 42
 - String Parameters 42

Port Parameters	43
Expression Actor	43
State Machines	45
5.4. Composite Data Types	45
5.4.1. <i>Arrays</i>	45
5.4.2. <i>Matrices</i>	47
5.4.3. <i>Records</i>	48
5.5. Invoking Methods in Expressions	50
5.6. Defining Functions in Expressions	51
5.7. Built-In Functions	53
5.8. Fixed Point Numbers	57
Appendix A: Tables of Functions	59
A.1. Trigonometric Functions	59
A.2. Basic Mathematical Functions	60
A.3. Matrix, Array, and Record Functions.	62
A.4. Functions for Evaluating Expressions	63
A.5. Signal Processing Functions	64
A.6. I/O Functions and Other Miscellaneous Functions	66
Appendix B: References	67

1. Introduction

VisualSense is a modeling and simulation framework for wireless and sensor networks that builds on and leverages Ptolemy II. Modeling of wireless networks requires sophisticated modeling of communication channels, sensors, ad-hoc networking protocols, localization strategies, media access control protocols, energy consumption in sensor nodes, etc. This modeling framework is designed to support a component-based construction of such models. It supports actor-oriented definition of network nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems. The software architecture consists of a set of base classes for defining channels and sensor nodes, a library of subclasses that provide certain specific channel models and node models, and an extensible visualization framework. Custom nodes can be defined by subclassing the base classes and defining the behavior in Java or by creating composite models using any of several Ptolemy II modeling environments. Custom channels can be defined by subclassing the `WirelessChannel` base class and by attaching functionality defined in Ptolemy II models. It is intended to enable the research community to share models of disjoint aspects of the sensor nets problem and to build models that include sophisticated elements from several aspects.

In this document, we describe a specialization of the discrete-event domain of Ptolemy II supporting sensor nets modeling. We begin by explaining the basic components in this framework: the director, the channel model and the sensor node model, and how to build sensor network models graphically. We then progress to discuss the software architecture of VisualSense, and how to extend the software for customized node models and channel models. This document provides a tutorial that will enable the reader to construct elaborate sensor network models and to have confidence in the results of a simulation of those models.

The intended audience for this document is an engineer or researcher who is interested in wireless and sensor network systems and wishes to build models of such systems.

VisualSense is built on top of Ptolemy II, a framework supporting the construction of such domain-specific tools. See <http://ptolemy.eecs.berkeley.edu> for information about Ptolemy II.

1.1 Installation and Quick Start

VisualSense can be quickly downloaded and run using Web Start¹, a standard Windows installer, or source code from the web site:

```
http://ptolemy.eecs.berkeley.edu/visualsense
```

Once you have done this, you can select *VisualSense* from the *Ptolemy II* entry in the Start menu (if you are using a Windows system). VisualSense can also be invoked from the command line on all platforms using the command:

```
vergil -visualsense
```

1. Web Start is a tool from Sun Microsystems that makes software installation and updates particularly simple. The Web Start installation works best with Windows, but has also been tried under Solaris, Linux and Mac OS X. The Web Start installation behaves almost exactly like a standalone installation; you can save models locally, and you need not be connected to the net after the initial installation. The Web Start tool includes a Java Runtime Environment (JRE), and the VisualSense Web Start installer checks that the proper version of the JRE is present.

You should then see an initial welcome window that looks like the one in figure 1. Feel free to explore the links in this window. To create a new model, invoke the New command in the File menu. But before doing this, it is worth understanding how a model works.

2. Modeling Wireless Networks

In this section, we explain how to read, construct and execute models of wireless sensor networks. We begin by examining a demonstration system that is accessible from the welcome window in figure 1, the wireless sound detection model. These demonstration systems are meant to illustrate capabilities, not necessarily to serve as accurate or useful models of physical systems.

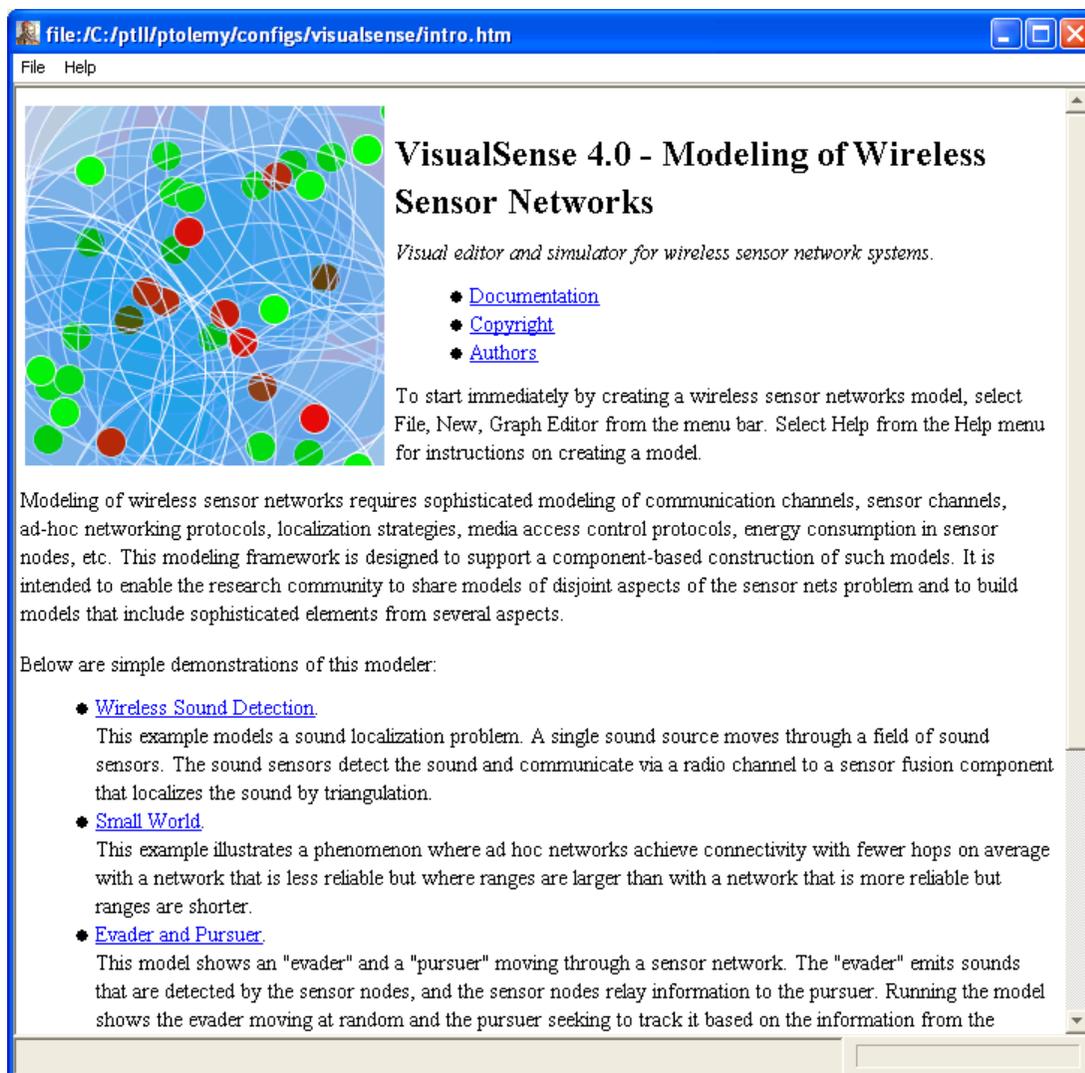


FIGURE 1. Initial welcome window for VisualSense.

2.1 Running a Pre-Built Model

The wireless sound detection model can be accessed by clicking on the link in the welcome window (figure 1), which results in the window shown in figure 2. This is a highly simplified (even naïve) model of a sound localization system that uses a field of sensor nodes that detect a sound and report by radio to a hub that triangulates the location of the sound. Figure 2 shows the elements of the model, which include a WirelessDirector, which defines this as a wireless model, two channel models (a radio channel model and a sound channel model), a number of annotations (text explaining the model) and *actors* in the model. Each of these components plays a role in the model. The director mediates execution of the model. The channel models handle communication between the actors. The actors send and receive signals via the channel.

The model is executable. Clicking on the red triangle in the toolbar results in the SoundSource actor (represented by concentric transparent circles) beginning to move in a circular pattern, as indicated by the blue arrow in figure 3. The SoundSource actor emits events via the SoundChannel channel model. These events propagate with a time delay dependent on distance to the blue circular nodes. When these nodes detect the sound, they emit a radio signal via the RadioChannel model and turn their icons red to indicate visually that they have done so. The radio signals include a time stamp of the detected sound event. The Triangulator actor in the center (shown with a green icon) receives these

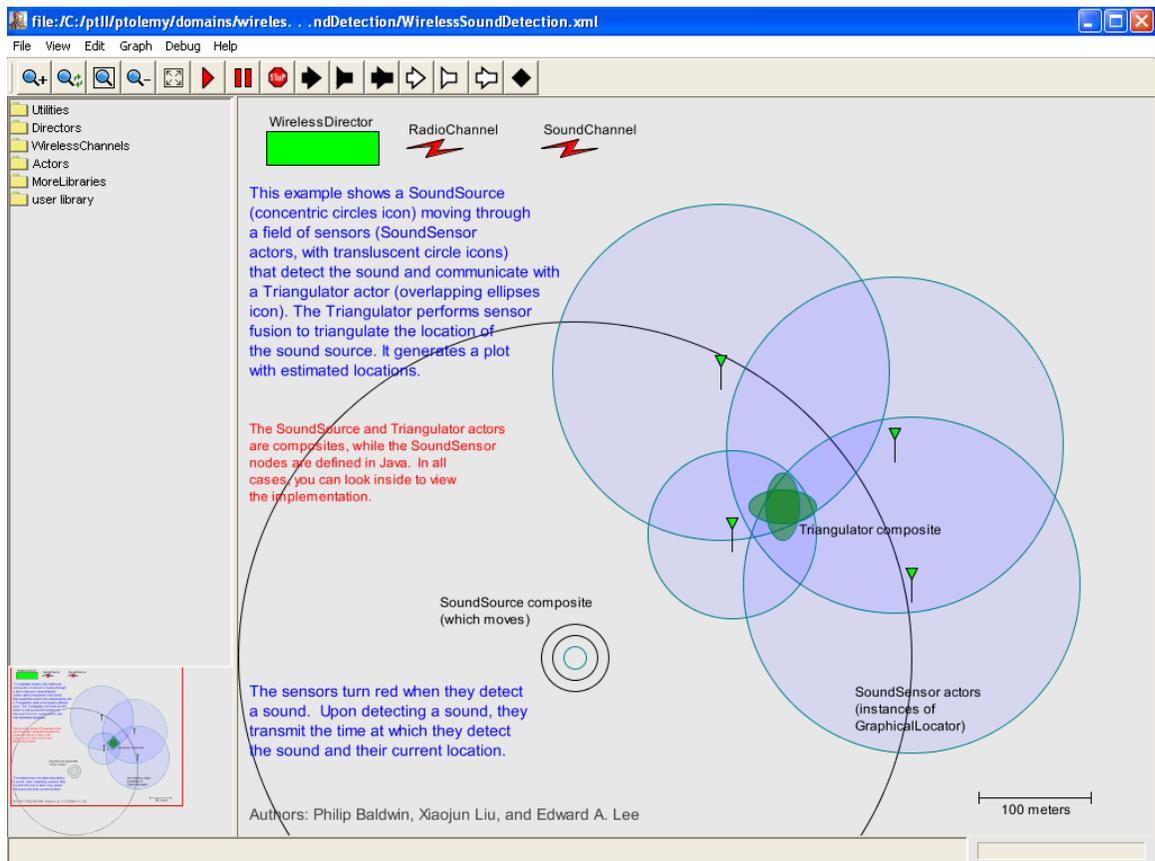


FIGURE 2. The VisualSense representation of a wireless sound detection model.

radio signals (if it is in range of the transmitter), and uses the time stamps to estimate the position of the sound source. It then plots that position, resulting in the plot shown in figure 3.

2.2 Changing Parameters

The model has parameters that you can experiment with. The parameters of two components, SoundSource and SoundChannel, are shown in figure 4. To obtain these parameter screens, you can double click on the actor, or right click and select “Configure.” The SoundSource has a single parameter, called *soundRange*. If you change the value from 300 (meters) to, say, 500, then the circular icon for the actor increases in size, and re-running the model results in more of the trajectory of the sound source being triangulated. In the SoundChannel parameters, you could set a non-zero value for the *lossProbability*, in which case only some of the sound events will be detected. Setting the *seed* to a non-zero value results in repeatable experiments, meaning that each execution will yield the same sequence of random numbers (the type is a *long*, so the value should be an integer followed by the letter “L”). Leaving the *seed* at the default “0L” yields a new experiment on each run.

2.3 Structure of a Pre-Built Model

Let us examine how the model in figure 2 is constructed.

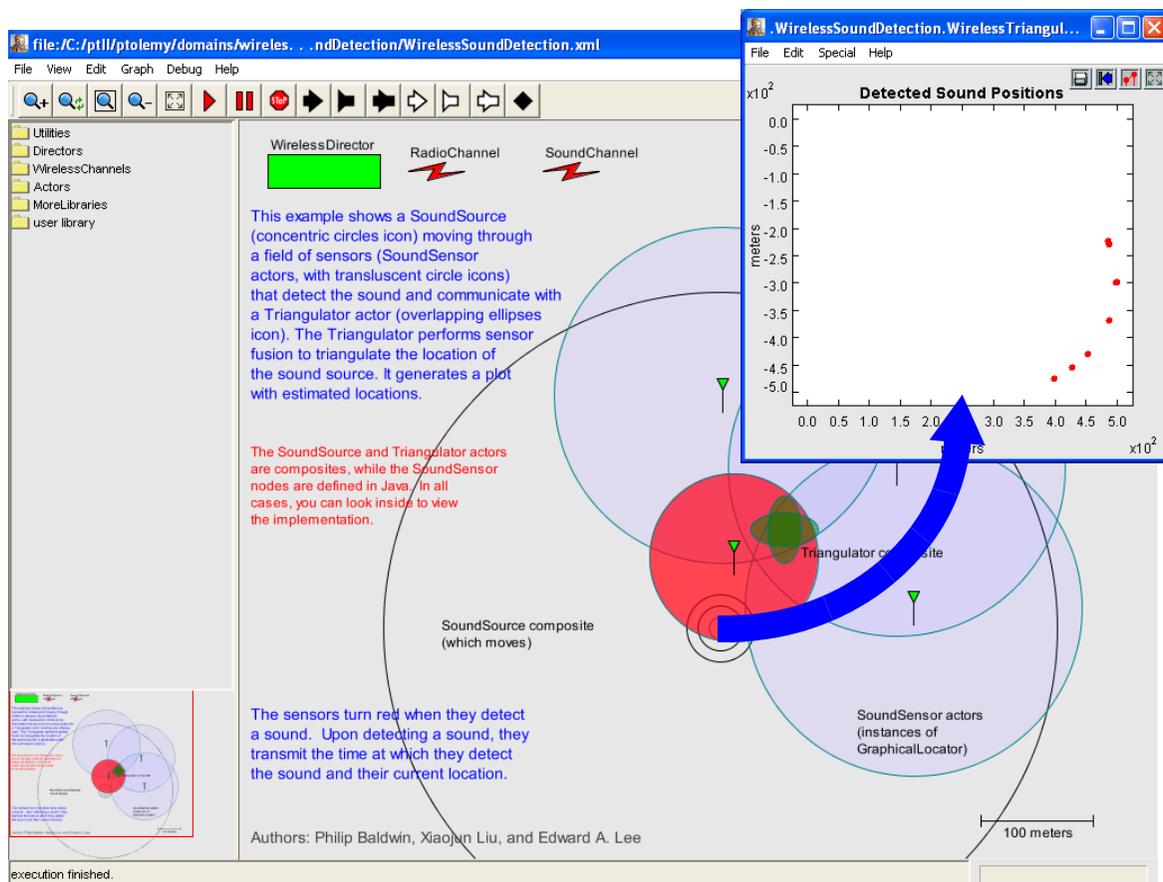


FIGURE 3. Animation as the model executes. The SoundSource actor moves in a circle through a field of Sound-Sensor actors. When these actors detect a sound, they transmit a radio signal to a Triangulator node, which estimates and plots (at the upper right) the position of the sound source.

2.3.1 Visual Representations (Icons)

Consider first the SoundSource actor. First, consider how its visual representation (its “icon”) changed when we changed the *soundRange* parameter. The definition of the icon can be viewed (and edited) by right clicking on the icon and selecting “Edit Custom Icon.” Note that to select this actor, you must place the mouse over one of the concentric circle outlines. The resulting window is shown in figure 5. Note that only the center portion of the icon is visible. Click on Zoom Fit in the toolbar (as shown in figure 5) to get the full image, as shown in figure 6. The navigation window at the lower left can be used to move the view around (to “pan” the view). The library at the left can be used to add items to the icon.

Consider the outer circle, which changed size when we changed the *soundRange* parameter. Double clicking on it (or right clicking and selecting Configure) reveals the parameter window in figure 7. Notice that the *width* and *height* parameters are given by expressions with values “ $\text{soundRange} * 2$ ”. The expression language that can be used here is rich, and will be described below. For now, it is sufficient to realize that arithmetic expressions that reference parameters of the actor or of the model can be used to extensively customize the visual representation of an actor, making it depend on parameter values.

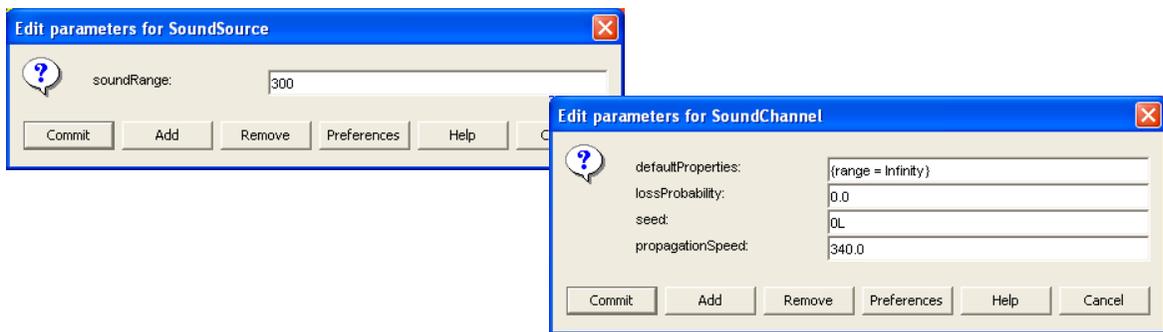


FIGURE 4. Parameters of the SoundSource actor (left) and SoundChannel channel model (right).

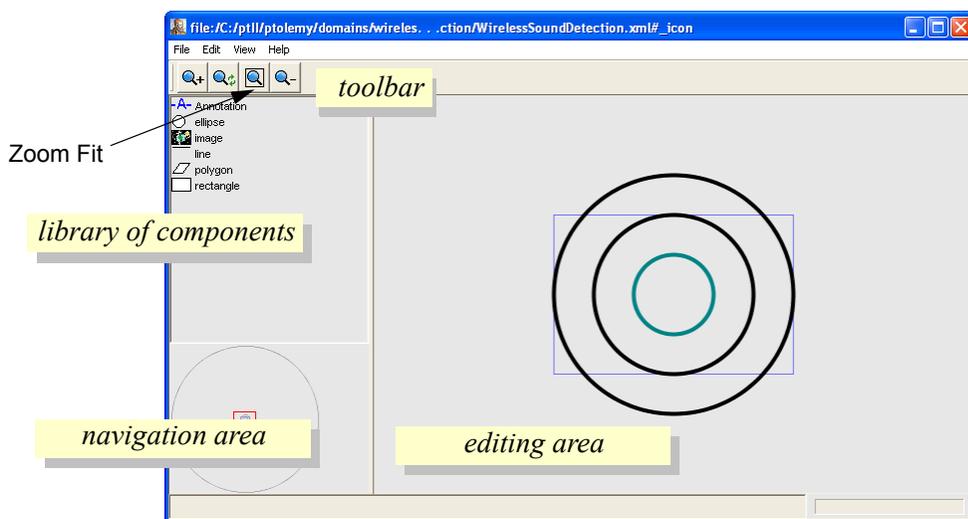


FIGURE 5. View resulting from selecting “Edit Custom Icon” after right clicking on the SoundSource in figure 2.

For example, we could fill the outer circle with a translucent color where the degree of translucency depends on the *soundRange* parameter, as shown in figure 8. In that figure, the color selector (shown at the right) was used to select a red color, and the *alpha* value of the color, which is the fourth element of the array defining the color, was manually set to “*soundRange/1000.0*”. The result is shown in figure 9.

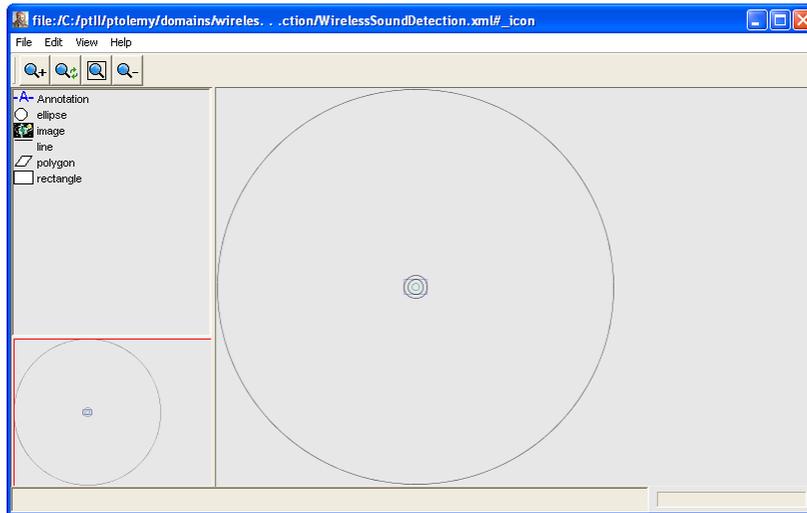


FIGURE 6. View resulting from clicking Zoom Fit in the toolbar of figure 5.

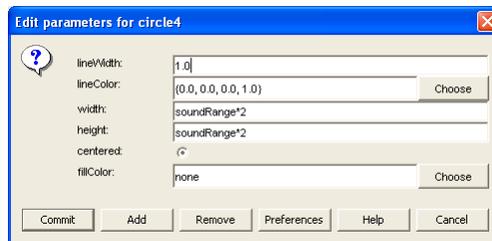


FIGURE 7. Parameters of the outer circle of the SoundSource actor icon in figure 5.

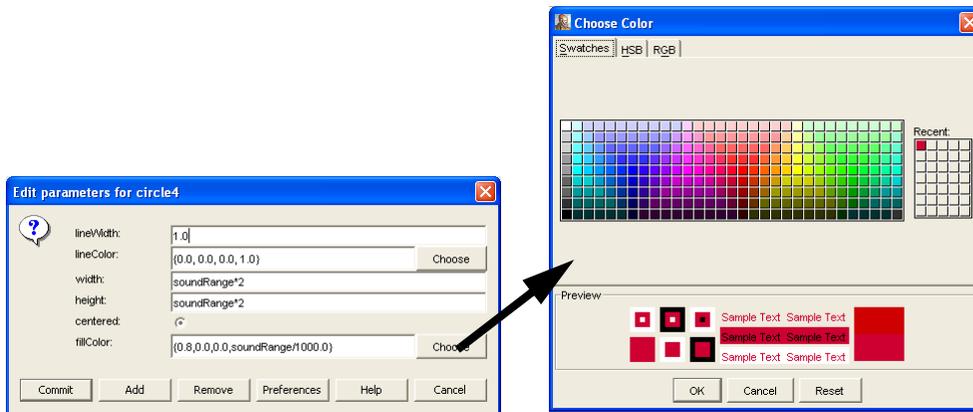


FIGURE 8. Setting the fill color of the outer circle of SoundRange to depend on its soundRange parameter.

Feel free to experiment with this icon by moving components, changing their colors, or adding new components. You can add GIF or JPEG images defined in a file using the Image component, and you can add lines, circles, polygons, or rectangles.

Note that as of this writing, the icon editor is fairly primitive. The interactors for the various shapes are not customized, so defining a shape can be a tedious matter of defining the vertex points. Also, the order in which items in the icon are drawn is the order in which they are created. Thus, the only mechanism currently to put an object in the foreground is to select it, delete it, and then re-add it. We expect this editor to improve over time.

2.3.2 Channels

The model shown in figure 2 has two channel models, shown in figure 10 along with their parameters. You can see that the only difference between these two channels (besides their names) is the value of the *propagationSpeed* parameter. For the RadioChannel, it is set to “Infinity,” whereas for the SoundChannel, it is set to “340.0” (meters/second).

Note that both channels have a parameter called *defaultProperties* with value “{range=Infinity}.” This expression defines a *record* with one field named “range” with value “Infinity.” The fields of the *defaultProperties* parameter of a channel define the ways in which a particular transmission can be individually customized. In this case, a particular transmission through either channel can optionally specify a range. If it is not specified, then the default is used, which is Infinity, indicating that there is no range limitation. A transmission will succeed in reaching the receiver no matter how far away the receiver is.

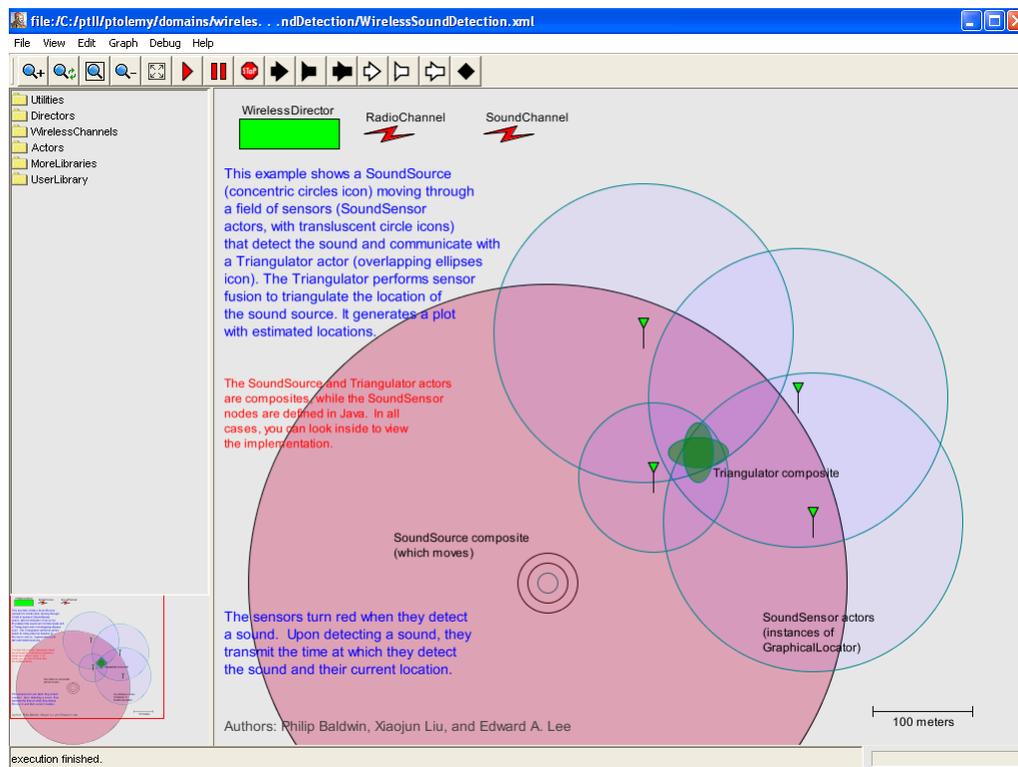


FIGURE 9. Result of changing the color of the outer circle of SoundRange as shown in figure 8.

2.3.3 Composite Actors

We have seen how to customize the visual representation of an actor. How can we define its behavior? The SoundSource actor in figure 2 is actually a *composite actor* whose behavior is defined by a Ptolemy II model. To find this definition, simply right click on the actor and select Look Inside. The inside model is shown in figure 11.

The SoundSource composite shown in figure 11 has a DEDirector (a discrete event director), which defines this model as a Ptolemy II discrete event model. DE models work well with wireless models, so it is common to see DE models used to define wireless nodes. The *soundRange* parameter is shown next to the DEDirector with its default value, 300. The model itself consists of two parts, an upper part that sends a sound event, and a lower part that moves the icon.

Consider first the upper part. It has a Clock and a *port* named “soundPort,” as shown in figure 12. The parameters of both the Clock and the port are obtained by double clicking on them (or right clicking and selecting Configure), and are also shown in the figure. Notice that the *period* of the Clock is set to 2.0, and the *values* are set to {1}, an array with one element, the integer 1. This indicates that the

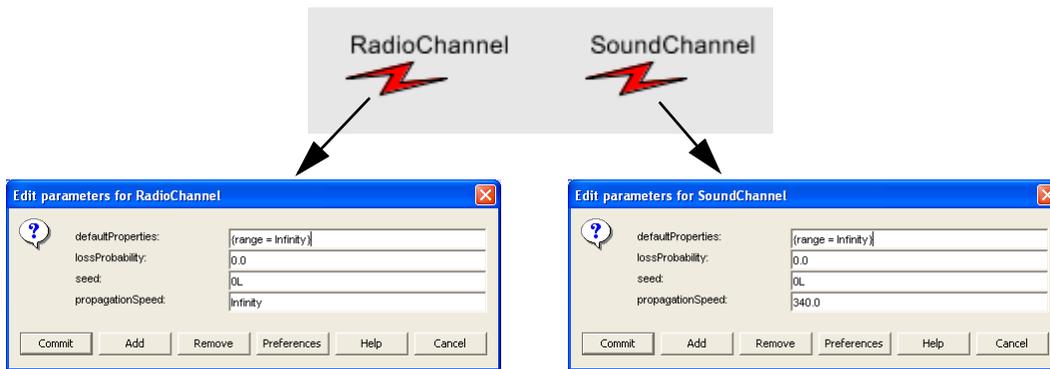


FIGURE 10. The channels of figure 2 and their parameters.

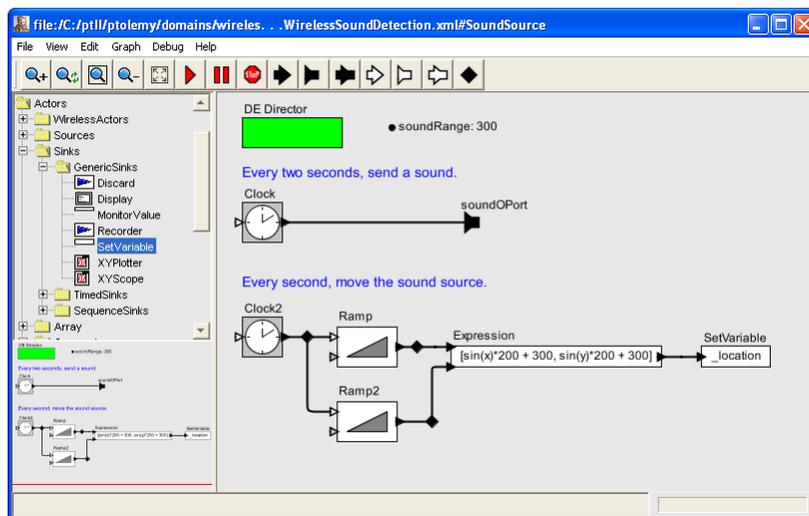


FIGURE 11. Result of looking inside the SoundRange actor in figure 2.

clock should produce a sound every two seconds. The value produced is simply the integer 1, which has no particular meaning. Any value would have the same effect.

The *soundPort* component also has parameters, as shown in figure 12. The *outsideChannel* parameter is a string-valued parameter with value “SoundChannel.” This is the name of the channel that this port will use for transmission, and must correspond with the name of the channel shown in figure 10. The *outsideTransmitProperties* parameter has value “{range=soundRange}” which is a *record* with one field named “range” with value given by the expression “soundRange,” which simply obtains the value from the *soundRange* parameter of the composite actor. Notice that this will override the default value of Infinity given for this field in figure 10. Thus, the *soundRange* parameter controls not just the visual appearance of the icon, but also the range of transmission.

For the purposes of determining whether a receiver is in range, all of the demos included with VisualSense use the location of the icon as a (two dimensional) representation of the location of the node. The units are arbitrary, but in these models are taken to represent meters. A scale is shown at the lower right of figure 2, indicated by a line of length “100,” which represents 100 meters.

Although these demos all use two-dimensional locations, the underlying software infrastructure supports three dimensional locations. The visual editor, however, does not offer a mechanism for directly defining those locations, so for illustration purposes, the demos constrain themselves to two-dimensional locations.

2.4 Controlling the Execution

The WirelessDirector in figure 2 is the component that controls the execution of the model. As with most components, it too has parameters. Its parameters are shown in figure 13. Notice that the

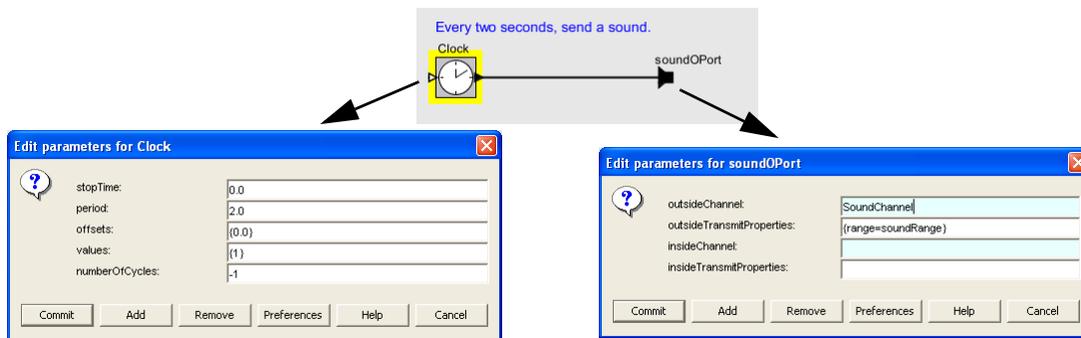


FIGURE 12. Portion of the composite in figure 11 that produces the sound event, with two parameter screens.



FIGURE 13. Parameters of the WirelessDirector of figure 2.

stop time is set to “MaxDouble,” which is a very large number $1.7976931 \times 10^{308}$. This specifies that the model should run forever.

Notice also that the *synchronizeToRealTime* parameter of the director is checked. This means that when executing the model, the Clock actor that produces a sound every two seconds will not be allowed to produce events at a faster rate than that in real time even if the model can execute faster. This parameter is used to get realistic time scales when animating an execution. Usually, this parameter should be checked for animated models. The other director parameters have to do with tuning the performance of the discrete-event simulator. They are beyond the scope of this document.

2.5 Building a New Model

We now proceed to build a new wireless network model from scratch. In any VisualSense window, select File→New→Graph Editor. This results in a window like that shown in figure 14. It contains a WirelessDirector, but nothing else. Drag in a PowerLossChannel from the WirelessChannels library at the left, as shown in figure 15.

Notice the parameters of this channel, which are also shown in figure 15. Notice that the *default-Properties* parameter contains a record with two fields, {range = Infinity, power = Infinity}. This channel can be used to model variations in transmit power and also power loss as a function of distance. We will construct a simple model that achieves communication if the receiver gets enough power, and does not achieve communication otherwise.

Documentation for the PowerLossChannel actor (and any other actor) can be obtained by right clicking on the actor and selecting Get Documentation. In this example, we get the screen shown in figure 16, which shows automatically generated documentation for the Java class that defines this channel. The top of this display shows the inheritance chain for the actor, which indicates that this actor extends LimitedRangeChannel, which extends DelayChannel, which extends ErasureChannel, which extends AtomicWirelessChannel. Each of these channels adds a small amount of functionality, and source code for each one is provided as an illustration of how to define channel models. You can view the source code by right clicking and selecting Look Inside (assuming you have installed the source code module), which results in the screen shown in figure 17. In the case of both the source

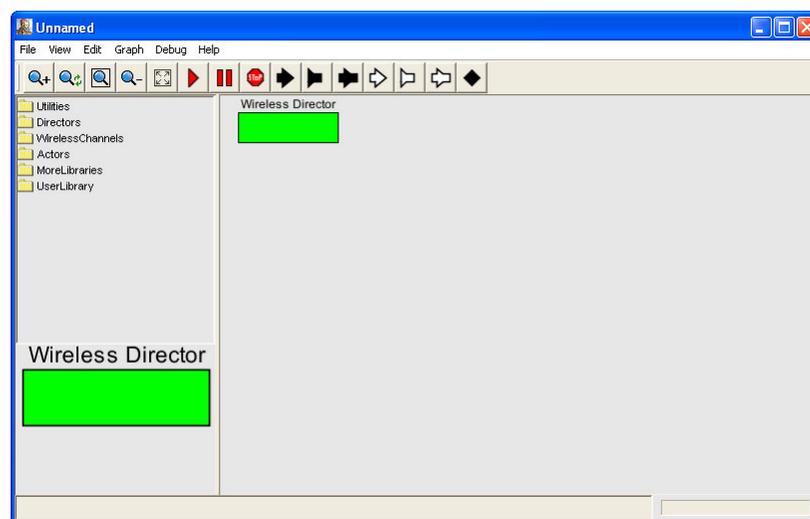


FIGURE 14. Window for constructing a new model, obtained from the menu File→New→GraphEditor.

code and the documentation, you have to scroll down some to get to the interesting part. For example, this documentation explains the *powerPropagationFactor* parameter as follows:

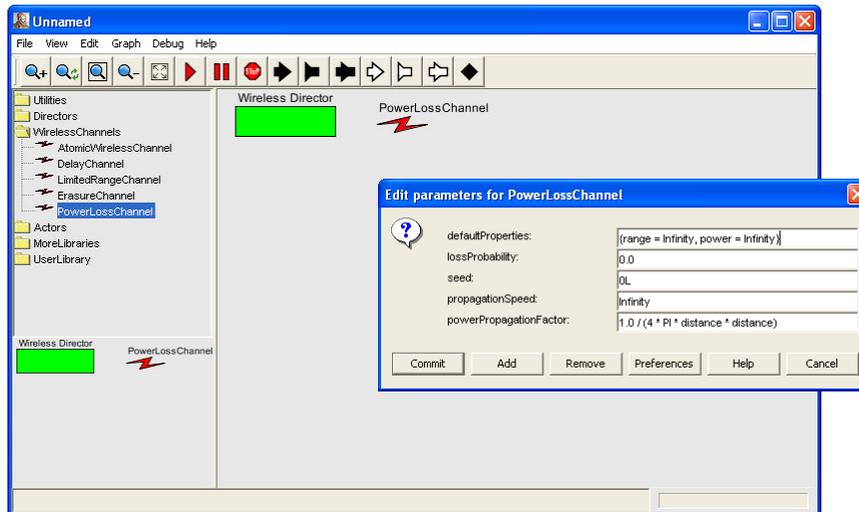


FIGURE 15. New model populated with a channel.

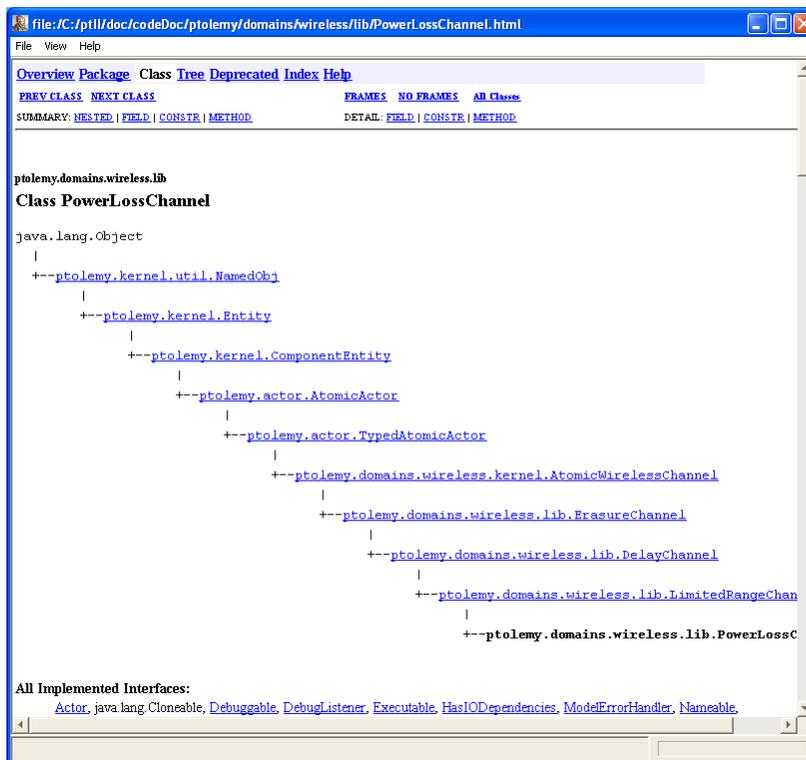


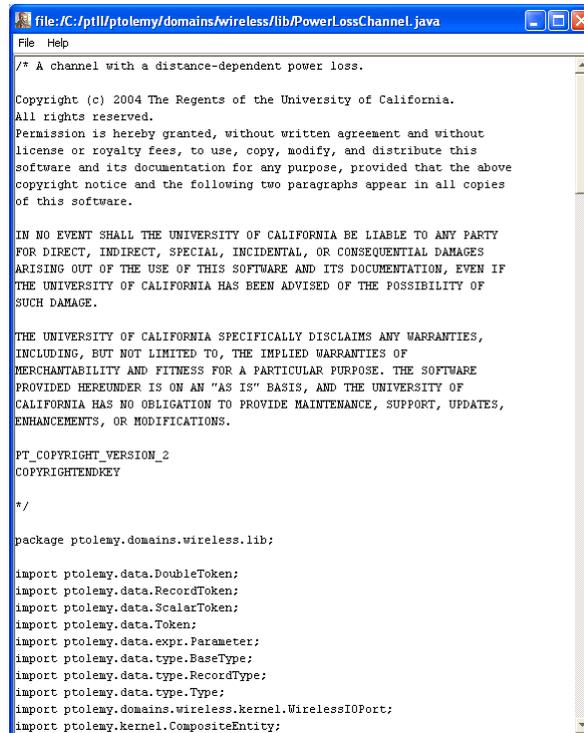
FIGURE 16. Documentation window for the PowerLossChannel, obtained with right click, Get Documentation.

“The power propagation is given as an expression that is evaluated and then multiplied by the power field of the transmit properties before delivery to the receiver. For convenience, a variable named “distance” is available and equal to the distance between the transmitter and the receiver when the power propagation formula is evaluated. Thus, the expression can depend on this distance. The value of the power field should be interpreted as power at the transmitter but power density at the receiver. A receiver may multiply the power density with its efficiency and an area (typically the antenna area). A receiver can then use the resulting power to compare against a detectable threshold, or to determine signal-to-interference ratio, for example.

The default value of *powerPropagationFactor* is

$$1.0 / (4 * \text{PI} * \text{distance} * \text{distance}).$$

This assumes that the transmit power is uniformly distributed on a sphere of radius *distance*. The result of multiplying this by a transmit power is a power density (power per unit area). The receiver should multiply this power density by the area of the sensor it uses to capture the energy (such as antenna area) and also an efficiency factor which represents how effectively it capture the energy.



```
file:/C:/ptill/ptolemy/domains/wireless/lib/PowerLossChannel.java
File Help

/* A channel with a distance-dependent power loss.

Copyright (c) 2004 The Regents of the University of California.
All rights reserved.
Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

PT_COPYRIGHT_VERSION_2
COPYRIGHTENDKEY

*/

package ptolemy.domains.wireless.lib;

import ptolemy.data.DoubleToken;
import ptolemy.data.RecordToken;
import ptolemy.data.ScalarToken;
import ptolemy.data.Token;
import ptolemy.data.expr.Parameter;
import ptolemy.data.type.BaseType;
import ptolemy.data.type.RecordType;
import ptolemy.data.type.Type;
import ptolemy.domains.wireless.kernel.WirelessIOPort;
import ptolemy.kernel.CompositeEntity;
```

FIGURE 17. Source code for PowerLossChannel obtained by right clicking and selecting Look Inside.

The power field of the transmit properties can be supplied by the transmitter as a record with a power field of type double. The default value provided by this channel is Infinity, which when multiplied by any positive constant will yield Infinity, which presumably will be above any threshold. Thus, the default behavior is to encounter no power loss and no limits to communication due to power.”

Hopefully, this makes it reasonably clear how to use these parameters. Let us build a model that uses them.

Begin by dragging in two instances of *WirelessComposite* from the Actors→*WirelessActors* library at the left. Rename them Transmitter and Receiver by right clicking on them and selecting *Customize Name*, to get the result shown in figure 18. These components now need ports. To create these, right click on each icon and select *Configure Ports*. Click on the Add button and create an output port named *output* for the Transmitter, and an input port named *input* for the Receiver, as shown in figure 19. To specify that these ports use the *PowerLossChannel*, right click on each port and select *Configure*, and specify the *outsideChannel* to be “*PowerLossChannel*” (this must match exactly the name of the channel).

We start by populating the transmitter and receiver with simple models of the nodes. To do this, look inside the transmitter, which yields the window shown in figure 20. Note that the output port is (rather poorly) placed at the upper left. Move it to a more reasonable place, and connect to it an instance of the *PoissonClock* actor from the Actors→*Sources*→*TimedSources* library to get the model shown in figure 21. To make a connection, either click and drag from the output port of the *PoissonClock* actor, or control-click and drag from output port of the Transmitter to the output port of the *PoissonClock* actor.

The *PoissonClock* actor will produce events at random times, where the time between events is obtained from an exponential random variable with mean given by the *meanTime* parameter of the *PoissonClock*. The default value is 1.0, which is fine for our purposes. If you return to the top-level window and double click on the *WirelessDirector* to set its *synchronizeToRealTime* parameter, then the transmitter will produce events at an average rate of one per second.

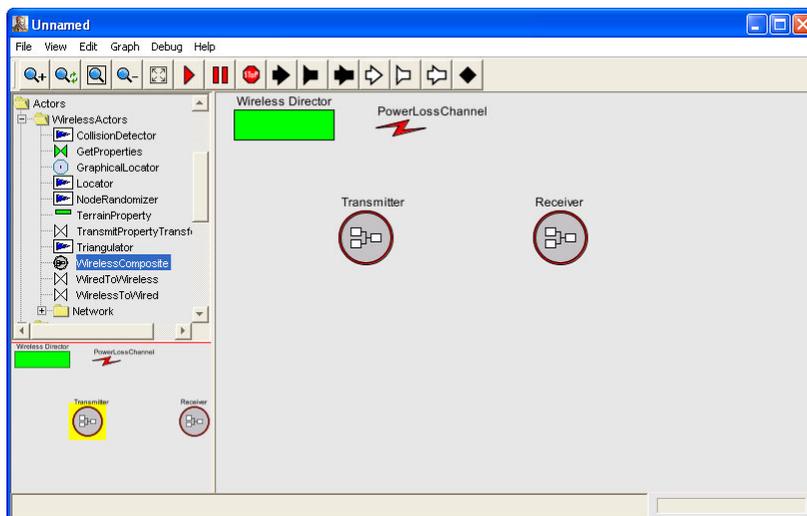


FIGURE 18. Model populated with two instances of *WirelessComposite* renamed Transmitter and Receiver.

Look inside the Receiver actor and build the model shown in figure 22. The Ramp actor is found in the Actors library under Sources→SequenceSources, and the Display actor is found under Sinks→GenericSinks, as shown on the left in the figure. The model is now ready to execute. Clicking on the red triangle in the toolbar will result in the display shown in figure 23. The Ramp produces a count of arrivals. If you remembered to set the *synchronizeToRealTime* parameter of the WirelessDirector, then the count numbers will appear at random times with an average interval of one second.

You may want to save your model using the File→Save menu command. Use the file extension .xml (or .moml) to ensure that VisualSense will recognize this as a model file. Notice that the title bar on the window now reflects the name of your model, which is the same as the name of the file.

Let us modify this model so that the power loss of the channel as a function of distance is observed. To do this, find the GetProperties actor in the Actors→WirelessActors library, and replace

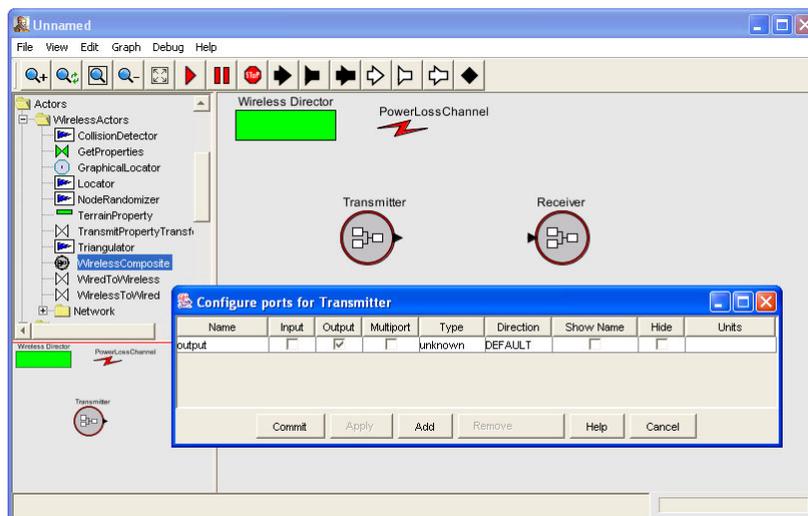


FIGURE 19. Model with ports added to the Transmitter and Receiver, and the dialog used to create the ports.

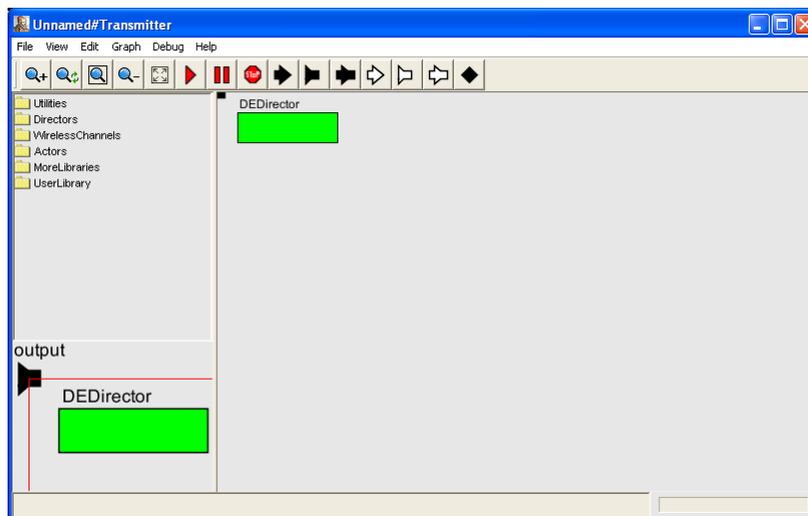


FIGURE 20. Inside the Transmitter.

the Ramp actor inside the Receiver as shown in figure 24. Running the model now results in the display shown in figure 25. Notice that the received power is always Infinity, which is not very useful.

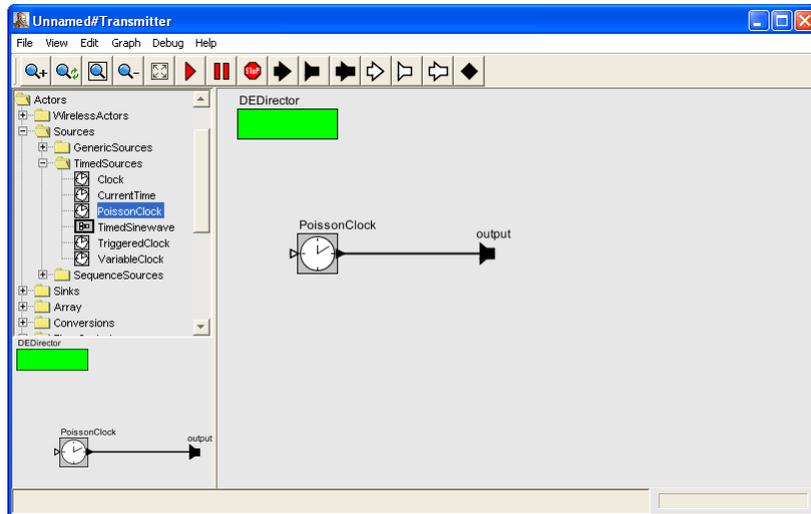


FIGURE 21. Completed Transmitter.

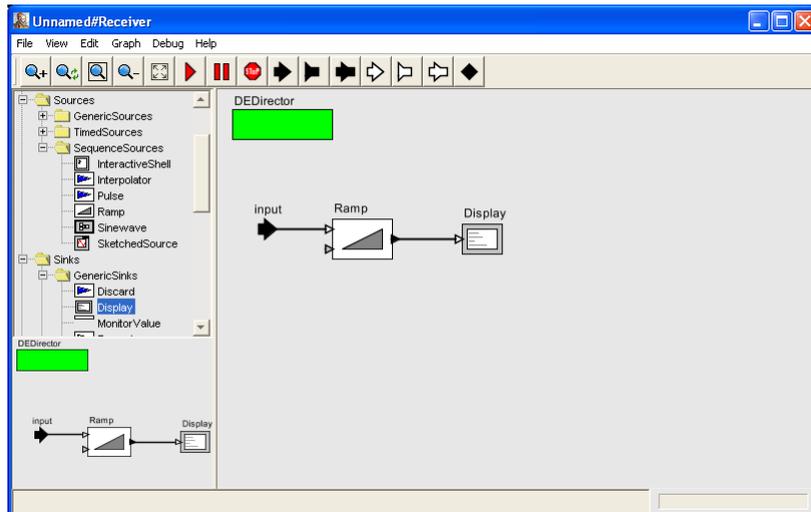


FIGURE 22. Completed Receiver.

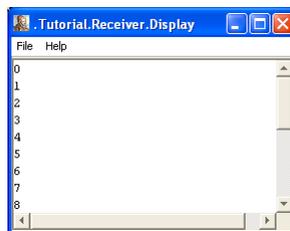


FIGURE 23. Display that results from running the model of figure 19.

Indeed, the Transmitter has not specified a transmit power, and the PowerLossChannel has a default power of Infinity, as shown in figure 15. The power loss introduced by the channel becomes irrelevant because in this model, the transmit power is infinite, which when multiplied by any non-zero loss, still yields infinite power.

To get a more reasonable model of power loss, set the transmit power by right clicking on the output port of the Transmitter and setting the *outsideTransmitProperties* parameter to “{power = 1.0;}” as shown in figure 26. Re-running the model now results in a display like that shown in figure 27, where the variability in power level was obtained by moving the Receiver towards and over the Transmitter while the model was running.

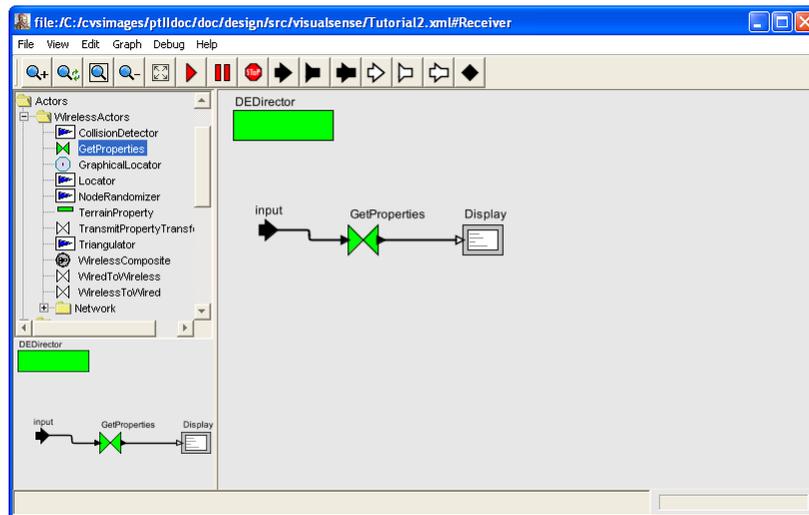


FIGURE 24. Modified Receiver that displays the received properties.

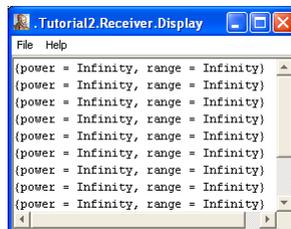


FIGURE 25. Display that results from using the Receiver design of figure 24.

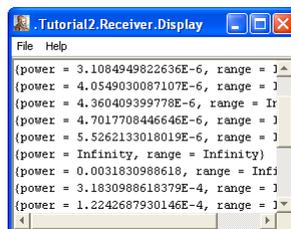


FIGURE 27. Display that results from using the transmit power set as shown in figure 26.

Notice in figure 27 that one of the displays shows a received power of Infinity. This occurred when the Transmitter and Receiver were directly on top of one another. Recall from the documentation for PowerLossChannel that the value of the *power* field in the received properties is a power density (power per unit area), not an absolute power. Hence, indeed, if the receiver and transmitter occupy the same physical space, and the transmitter is a point source, then the power density at the receiver is infinite. Typically, a receiver model will multiply this power density by an effective antenna area and an antenna efficiency to get an absolute received power level.

The received power density can be used to decide at the receiver whether transmission is successful. To do this, modify the Receiver model to get the structure shown in figure 28. The actors used here are found as follows:

- RecordDisassembler: Actors→FlowControl→Aggregators

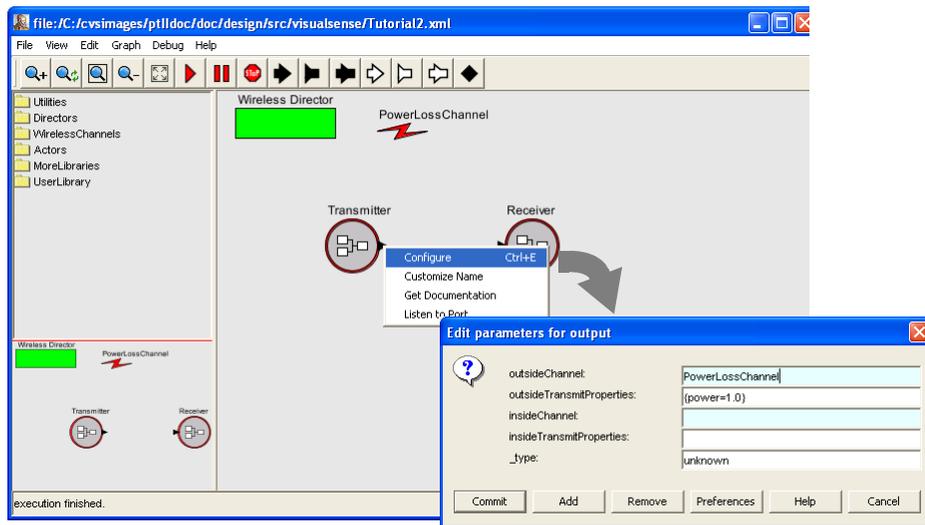


FIGURE 26. Setting the transmit power of the Transmitter.

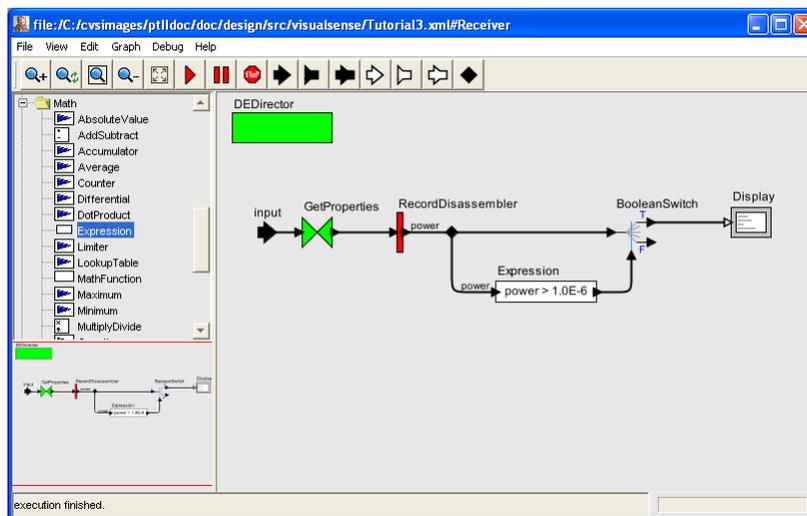


FIGURE 28. Receiver model that discards received events where the power is below a threshold.

- Expression: Actors→Math
- BooleanSwitch: Actors→FlowControl→BooleanFlowControl

The RecordDisassembler actor extracts fields from a record. To use it, you must create output ports that have the same name as the field, in this case, *power*. To use the Expression actor, you must create input ports, using whatever names you like (“*power*” in figure 28), and then give an expression that defines the output in terms of the inputs (“*power > 1.0E-6*” in figure 28). The output of this Expression actor will be *true* if the received power is greater than 1.0×10^{-6} , and false otherwise. That boolean signal drives the *control* port of the BooleanSwitch, which sends its input to one of two output ports depending on the value of the control input. In this case, we observe only the *true* output, which will be the received power values that exceed 1.0×10^{-6} .

Notice that in figure 28, some connections involve a small black diamond. This is the visual mechanism for routing a signal to multiple places. To create the diamond (which is called a *vertex*), you can either control click on the background of the editor, or click on the black diamond in the toolbar. To link wires to the vertex, hold the control key while clicking and dragging to draw the connection.

2.6 Using the Plot Actors

Often, it is more useful for a model to graph data rather than display it in textual form. Modify the model of figure 28 as shown in figure 29, where the Display actor has been replaced by a TimedPlotter from Actors→Sinks→TimedSinks. The result of a run is shown in figure 30, where the Receiver was moved during the execution so it passed very close to the Transmitter.

This plot display can be improved considerably. In the plot window, click on the format button at the upper right, as shown in figure 30, to get the window shown in figure 31. Setting the parameters as indicated in that window results in the plot in figure 32, which is a more appealing rendition of the data.

Notice that you can zoom into a region of the plot by simply clicking and dragging out the region of interest. You can zoom out by clicking and dragging upwards or leftwards rather than downwards or rightwards. You can zoom fit by clicking on the zoom fit button at the upper right.

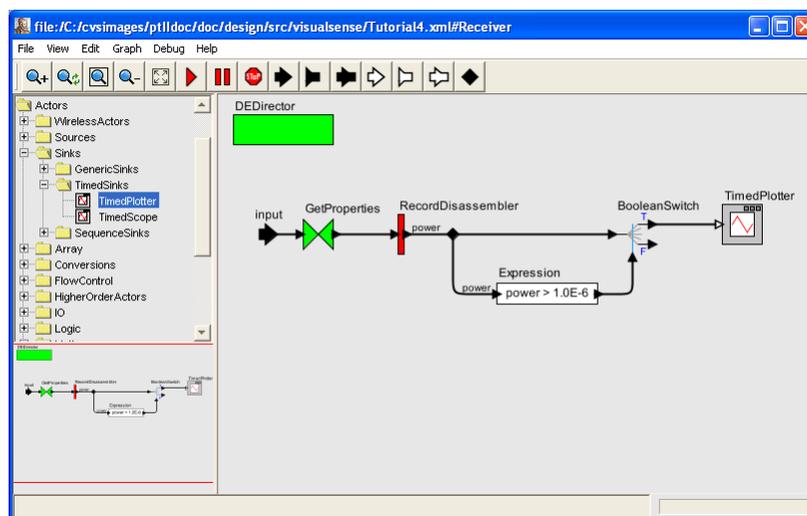


FIGURE 29. Receiver that plots rather than displays textually the received power as a function of time.

3. Modeling Capabilities

VisualSense is an extension of the discrete-event modeler of Ptolemy II. It largely preserves the discrete-event semantics, but changes the mechanism for connecting components so that explicit wires are not required. In the models constructed in the previous section, wired and wireless models were combined hierarchically. Indeed, all of Ptolemy II, which includes a very rich set of modeling mechanisms, can be used to construct very elaborate models of sensor nodes and propagation effects.

In this section, we explain the discrete-event semantics briefly and discuss the channel model that is used to decide connectivity in sensor nets and the hierarchical component model for each sensor node. We then illustrate capabilities by discussing some of the examples that are provided as demos with the system.

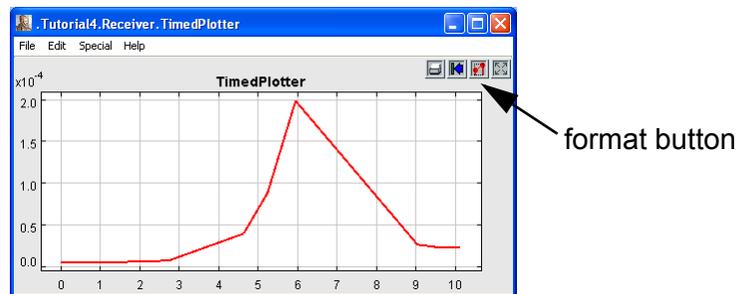


FIGURE 30. Plot showing the received power a function of time as the Receiver is moved close to the Transmitter.

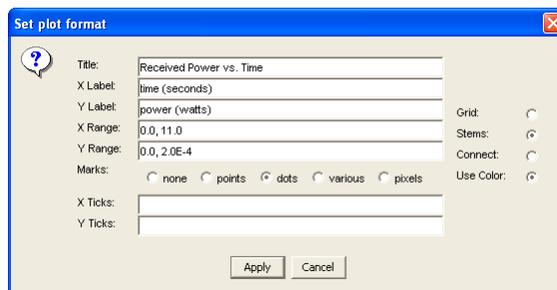


FIGURE 31. Dialog to set the plot format, filled in to yield the display shown in figure 32.

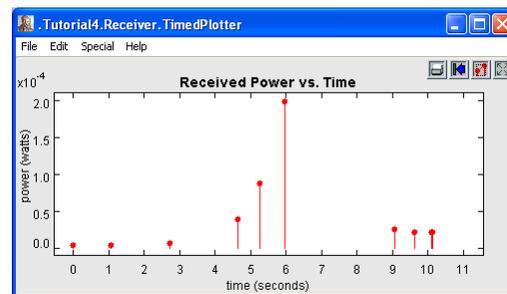


FIGURE 32. Plot display using the format shown in figure 31.

3.1 Discrete-Event Simulation

The director plays a key role in Ptolemy II: it defines the *semantics* of a composite. It gives the concurrency model and the communication mechanisms. In VisualSense, the director implements the simulator. The WirelessDirector is an almost completely unmodified subclass of the pre-existing discrete-event director (DEDirector) in Ptolemy II.

The discrete-event (DE) domain of Ptolemy II [10] provides execution semantics where interaction between components is via events with time stamps. The time stamps are double-precision floating point numbers, and a sophisticated calendar-queue scheduler is used to efficiently process events in chronological order. DE has a formal semantics that ensures determinate execution of deterministic models [12], although stochastic models for Monte Carlo simulation are also well supported. The precision in the semantics prevents the unexpected behavior that sometimes occurs due to modeling idiosyncrasies in some modeling frameworks.

The DE domain in Ptolemy II supports models with dynamically changing interconnection topologies. Changes in connectivity are treated as mutations of the model structure. The software is carefully architected to support multithreaded access to this mutation capability. Thus, one thread can be executing a simulation of the model while another changes the structure of the model, for example by adding, deleting, or moving actors, or changing the connectivity between actors. The results are predictable and consistent.

The most straightforward uses of the DE domain in Ptolemy II are similar to other discrete-event modeling frameworks such as NS, Opnet, and VHDL. Components (which are called actors) have ports, and the ports are interconnected to model the communication topology. Ptolemy II provides a visual editor for constructing DE models as block diagrams. However, such block diagrams are a poor representation of a sensor network, because the interconnection topology is highly variable.

VisualSense largely preserves DE semantics, but changes the mechanism for connecting components. In particular, it removes the need for explicit connections between ports, and instead associates ports with channels by name (e.g. “RadioChannel”). Connectivity can then be determined on the basis of the physical locations of the components. The algorithm for determining connectivity is itself encapsulated in a component as a channel model, and can be elaborated in the receiver models, and hence can be developed by the model builder.

3.2 Channel Models

A channel model in VisualSense is itself an actor. When a transmitter produces an event on a wireless port that references the channel by name, the event is delivered to the channel for transformation. The channel may alter the properties that are supplied by the transmitter, and may delay delivery of the event to a receiver to model propagation delay. In VisualSense, the responsibility of the channel ends there. Other components are used to model terrain effects, antenna gains, etc. Some of these are described below.

3.3 Wireless Node Models

Sensor nodes themselves can be modeled in Java, or more interestingly, using more conventional DE models (as block diagrams) or other Ptolemy II models (such as dataflow models, finite-state machines or continuous-time models). For example, a sensor node with modal behavior can be defined by sketching a finite-state machine and providing refinements to each of the states to define the behavior of the node in that state. This can be used, for example, to model energy consumption as a function

of state. Sophisticated models of the coupling between energy consumption and media access control protocols become possible.

3.4 Examples of Modeling Capabilities

Most of the modeling capabilities described here are illustrated in the quick tour, accessible from the welcome window shown in figure 1.

3.4.1 Packet Structure

Ptolemy II includes a sophisticated type system that includes aggregate types like records. Above, we showed how records can be used for transmit properties. They can also be used to construct packets with arbitrary payloads. The mechanisms are identical. The RecordAssembler, RecordDisassembler, and RecordUpdater actors in the Actors→FlowControl→Aggregators library can be used to assemble and disassemble records.

The type system will check for compatibility in uses of records. Extracting a field and using it incorrectly (e.g. using it as a boolean value when it is actually an integer) will yield a type check error before the model is executed.

3.4.2 Packet Losses

The ErasureChannel model, which is a base class for most of the channel models, offers a parameter *lossProbability* that can be used to model independent, identically distributed packet losses.

3.4.3 Battery Power

Since nodes in a wireless network can be defined by arbitrary Ptolemy models, it is easy to incorporate models of energy or power consumption. A simple example is given in the quick tour under “Circular Range Channel,” shown in figure 34, where on the right you can see that the Transmitter uses a PoissonClock to decrease the range of transmission at random times to model the transmission range degradation over time as its battery is depleted. When this model executes, the size of the circular icon representing the transmitter decreases as its range decreases.

3.4.4 Power Loss

The quick tour includes a model called “Power Loss Channel” that illustrates power variability at the receiver as a function of distance. The top-level model, receiver implementation, and a plot resulting from its execution are shown in figure 33. The model uses the same principles as the tutorial example described above.

3.4.5 Collisions

In the underlying discrete-event semantics of VisualSense, events occur instantaneously at a particular time. That is, they do not have a duration. To model collisions of messages that take time and share a common channel, the model must explicitly include the message duration.

A simple example of such a model is shown in figure 35. In this model, two transmitters share the same channel and transmit messages of fixed duration at random times. As the model executes, one of the transmitters moves in a circular pattern, starting far from the receiver, coming close, then moving away again. At the start, when it is far from the receiver, its messages get through to the receiver only if the other transmitter does not transmit a message that overlaps in time. Whether the message from the other transmitter gets through in the event of a collision depends on how far away the first transmit-

ter is. If it is sufficiently far away, then the interfering power is not sufficient to prevent communication, so the message gets through. If it is closer, then the interfering power will be sufficient that neither message gets through.

Two plots are shown in figure 35. The upper plot shows the messages that are transmitted (in red and blue), giving a visual indication of when overlap occurs. The magnitude in the plot represents the received power. For the transmitter that is stationary, the receiver power is constant. For the transmitter that moves, the received power starts low, then rises to nearly equal the power of the stationary transmitter, then drops again. The lower plot indicates whether messages are lost. In the figure, a total of seven messages are lost, all but one of them from the mobile transmitter (shown in red, if you have a color copy of this document).

The duration of a message in this model is represented by an extra field added to the transmit properties by the channel. The parameters of the channel are shown at the lower right in figure 35. Notice that the *defaultProperties* parameter has value “{range=Infinity, power=Infinity, duration=1.0}”. The *duration* field in this record represents the duration of a message. Individual transmitters can override this by setting the *outsideTransmitProperties* parameters of their ports to give any desired duration.

The Receiver implementation is shown in figure 36. In this model, the value of the received signal is a boolean with value *false* if the originator is the fixed transmitter and value *true* if the originator is the mobile transmitter. The *GetProperties* actor is used to extract the received properties, which will include the received power and the message duration. The *power* and *duration* fields of the properties

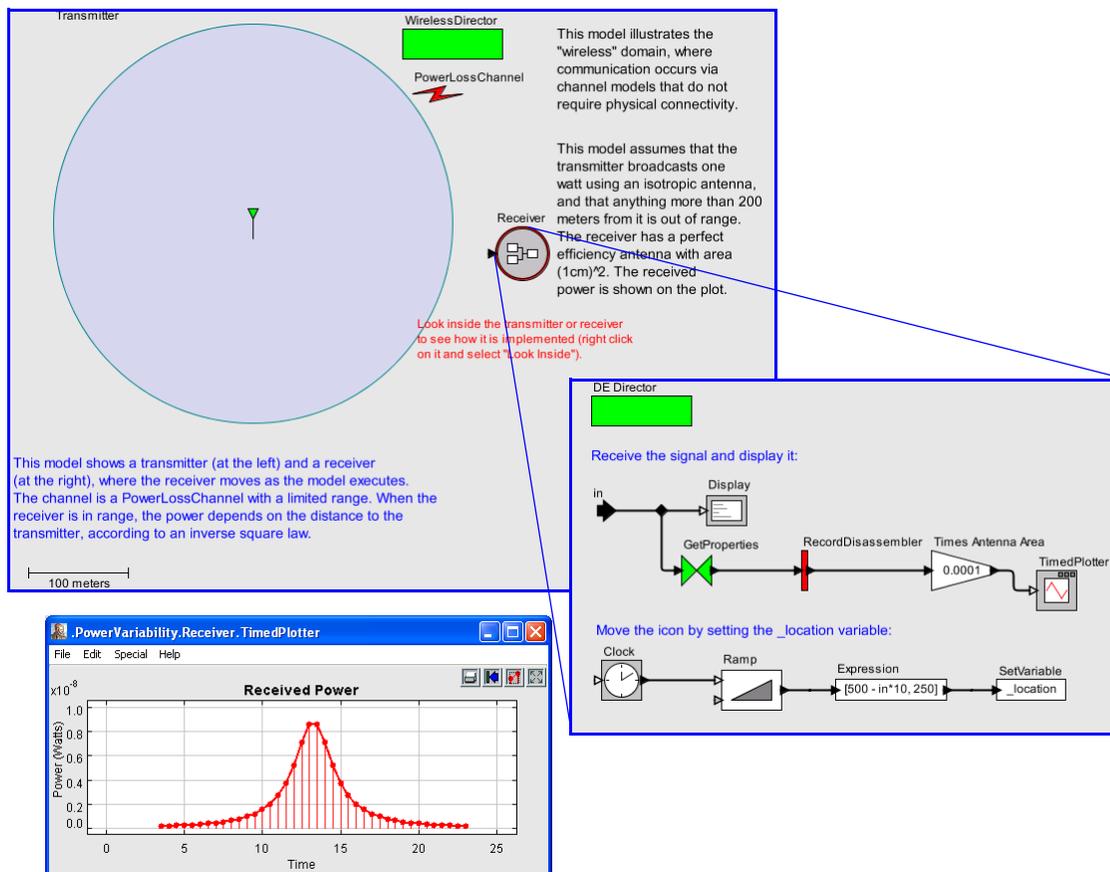


FIGURE 33. Model of power loss as a receiver moves into range and then close to a transmitter.

record are extracted by the RecordDisassembler actor and fed into the CollisionDetector actor, which determines which of the messages are received and which are lost. The rest of the model is devoted to constructing meaningful plots so that we get a visual rendition of the behavior.

The CollisionDetector actor is fairly sophisticated. Its documentation is shown in figure 37. This actor assumes that the duration of messages is short relative to the rate at which the actors move. That is, the received power (and whether a receiver is in range) is determined once, at the time the message starts, and remains constant throughout the transmission.

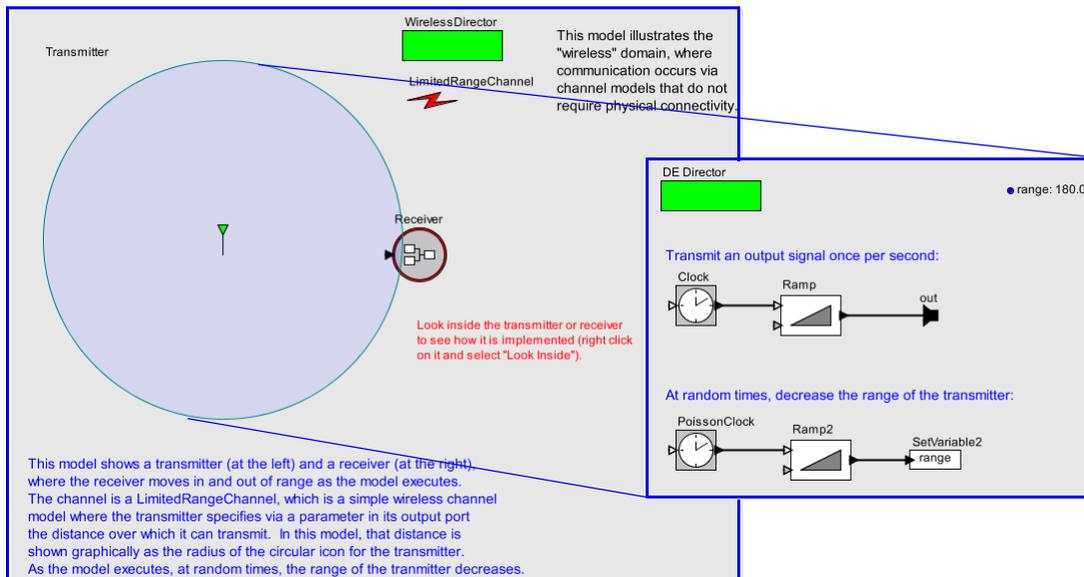


FIGURE 34. Model where transmission range degrades over time as a battery is depleted.

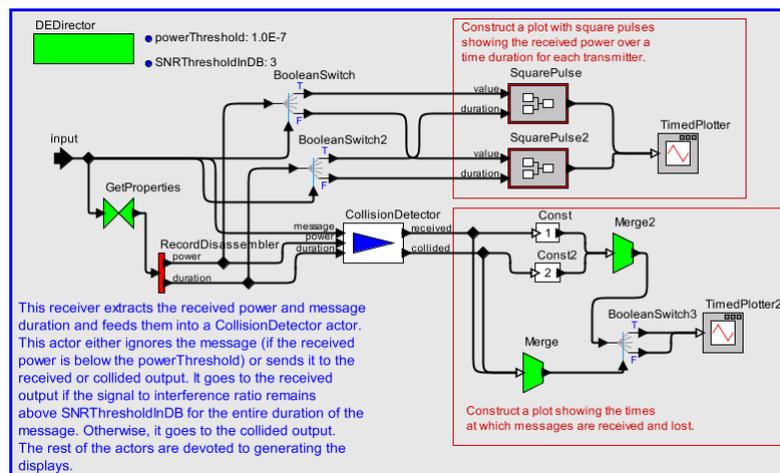


FIGURE 36. Implementation of the Receiver in figure 35, which models and tracks collisions.

3.4.6 Transmit Antenna Gain

A transmitter for a wireless channel may have a directional antenna. This introduces a significant complication in modeling because, although the directionality is a local property of the transmitter, its effect depends on the location of the receiver. We have seen above the use of transmit properties to model propagation losses. Transmit properties are also used to model antenna gains. The transmitter registers with the channel a *property transformer*, which is an actor that will modify the transmit properties for any particular transmission. Before the channel delivers an event to a receiver, it executes the property transformer, informing it of the location of the transmitter and receiver, and permitting it to modify the transmit properties.

An example of a model that includes a directional transmit antenna is shown in figure 38. This model is visible in the quick tour under “Transmit Antenna Gain.” When this model executes, the receiver moves in a circular pattern around the transmitter and measures and plots the received power. The transmitter has an 8-element phased-array antenna with steering.

The design of the transmitter is quite sophisticated, as is shown in figure 39. It illustrates how the full modeling power of Ptolemy II can be used in VisualSense. At the top left of the figure, the Trans-

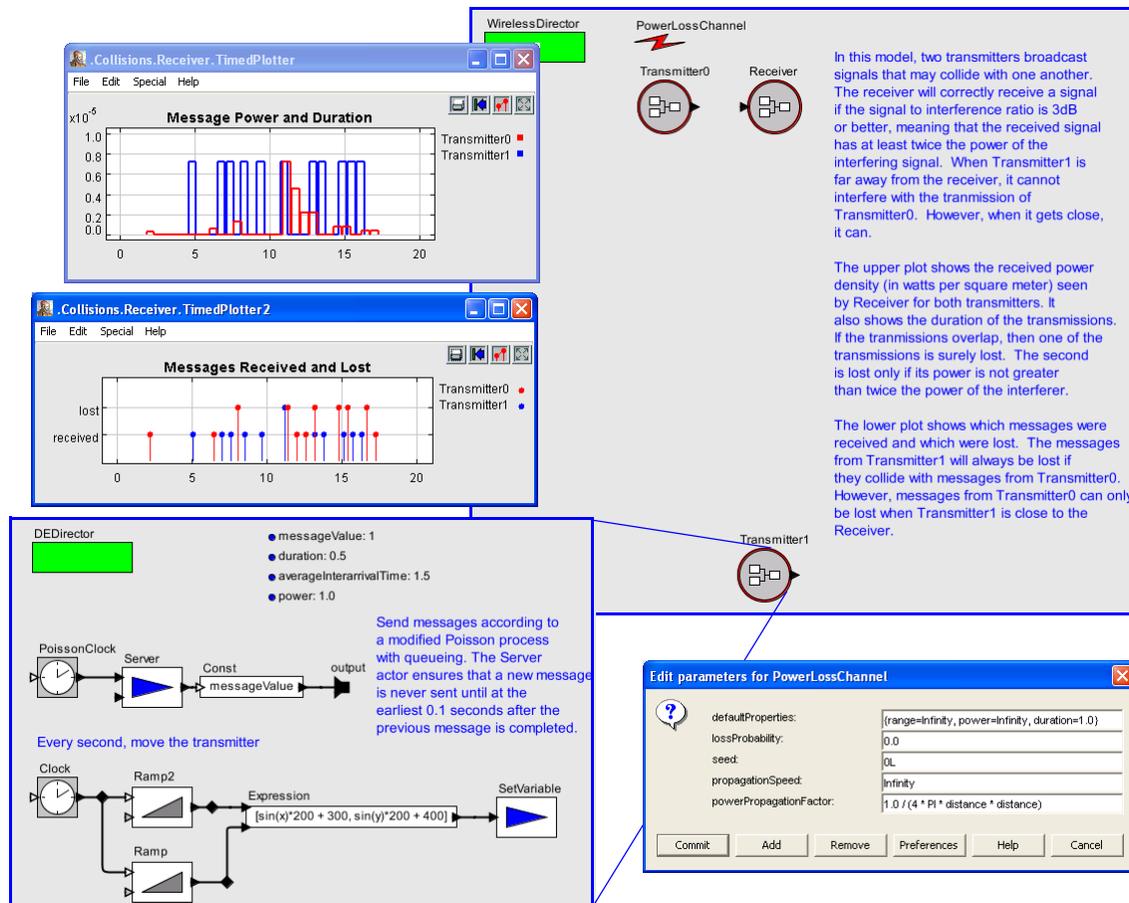


FIGURE 35. Model of collisions of messages that take time.

CollisionDetector: This actor models a typical physical layer front end of a wireless receiver. It models a receiver where messages have a non-zero duration and messages can collide with one another, causing a failure to receive. A message is provided to this actor at the time corresponding to the start of its transmission. Along with the message (an arbitrary token), the inputs must provide the duration of the message and its power. The message spans an interval of time starting when it is provided to this actor and ending at that time plus the duration. If another message overlaps with a given message and has sufficient power, then the given message will be sent to the *collided* output. Otherwise it is sent to the *received* output. In both cases, the message appears at the corresponding output at the time it is received plus the duration (i.e. the time at which the message has been completed).

The inputs are:

- *message*: The message carried by each transmission.
- *power*: The power of the received signal at the location of this receiver.
- *duration*: The time duration of the transmission.

The *power* and *duration* are typically delivered by the channel in the “properties” field of the transmission. The power is usually given as a power density (per unit area) so that a receiver can multiply it by its antenna area to determine the received power. It is in a linear scale (vs. DB), typically with units such as watts per square meter. The duration is a non-negative double, and the message is an arbitrary token.

The outputs are:

- *received*: The message received. This port produces an output only if the received power is sufficient and there are no collisions. The output is produced at a time equal to the time this actor receives the message plus the value received on the *duration* input.
- *collided*: The message discarded. This port produces an output only if the received message collides with another message of sufficient power. The output is produced at a time equal to the time this actor receives the message plus the value received on the *duration* input. The value of the output is the message that cannot be received.

This actor is typically used with a channel that delivers a properties record token that contains *power* and *duration* fields. These fields can be extracted by using a GetProperties actor followed by a RecordDisassembler. The PowerLossChannel, for example, can be used. However, in order for the type constraints to be satisfied, the PowerLossChannel's *defaultProperties* parameter must be augmented with a default value for the *duration*. Each transmitter can override that default with its own message duration and transmit power.

Any message whose power (as specified at the *power* input) is less than the value of the *powerThreshold* parameter is ignored. It will not cause collisions and is not produced at the *collided* output. The *powerThreshold* parameter thus specifies the power level at which the receiver simply fails to detect the signal. It is given in a linear scale (vs. DB) with the same units as the *power* input. The default value is zero, i.e. by default it won't ignore any received signal.

Any message whose power exceeds *powerThreshold* has the potential of being successfully received, of failing to be received due to a collision, and of causing a collision. A message is successfully received if throughout its duration, its power exceeds the sum of all other message powers by at least *SNRThresholdInDB* (which as the name suggests, is given in decibels, rather than in a linear scale, as is customary for power ratios). Formally, let the message power for the *i*-th message be $p_i(t)$ at time *t*. Before the message is received and after its duration expires, this power is zero. The *i*-th message is successfully received if

$$p_i(t) \geq P \sum_{j \neq i} p_j(t) \tag{1}$$

for all *t* where $p_i(t) > 0$, where $P = 10^{(SNRThresholdInDB/10)}$, which is the signal to interference ratio in a linear scale.

FIGURE 37. Documentation for the CollisionDetector actor used in figure 36.

mitPropertyTransformer actor models the transmitter antenna. Its firing behavior is very simple: when presented with an input token, it simply produces that same input token, unchanged, on the output port. However, in addition to this firing behavior, this actor registers itself with the channel used by the port that its output is connected to as a *property transformer*. When wireless communication occurs through that output port to some receiver, the channel calls back the TransmitPropertyTransformer once for each receiver, provides the location of the receiver, and executes the model contained by the TransmitPropertyTransformer actor.

The model contained by the TransmitPropertyTransformer actor is shown in figure 39. At the top right is the top level of this model. It shows that when it is executed (on request by the channel, once for each transmission), it is provided with three values, *senderLocation*, *receiverLocation*, and *properties*. The *properties* value is a record that in this case includes a *power* field that is to be modified by the model to account for the antenna gain in the direction from the transmitter to the receiver. This model calculates the angle of the transmission, calculates the antenna gain in that direction, and then scales the *power* field of the *properties* record. Notice that this model has an SDFDirector rather than the usual WirelessDirector or DEDirector used most commonly in VisualSense. This is because the calculation of antenna gain is essentially a signal processing function, something that the SDFDirector handles very well.

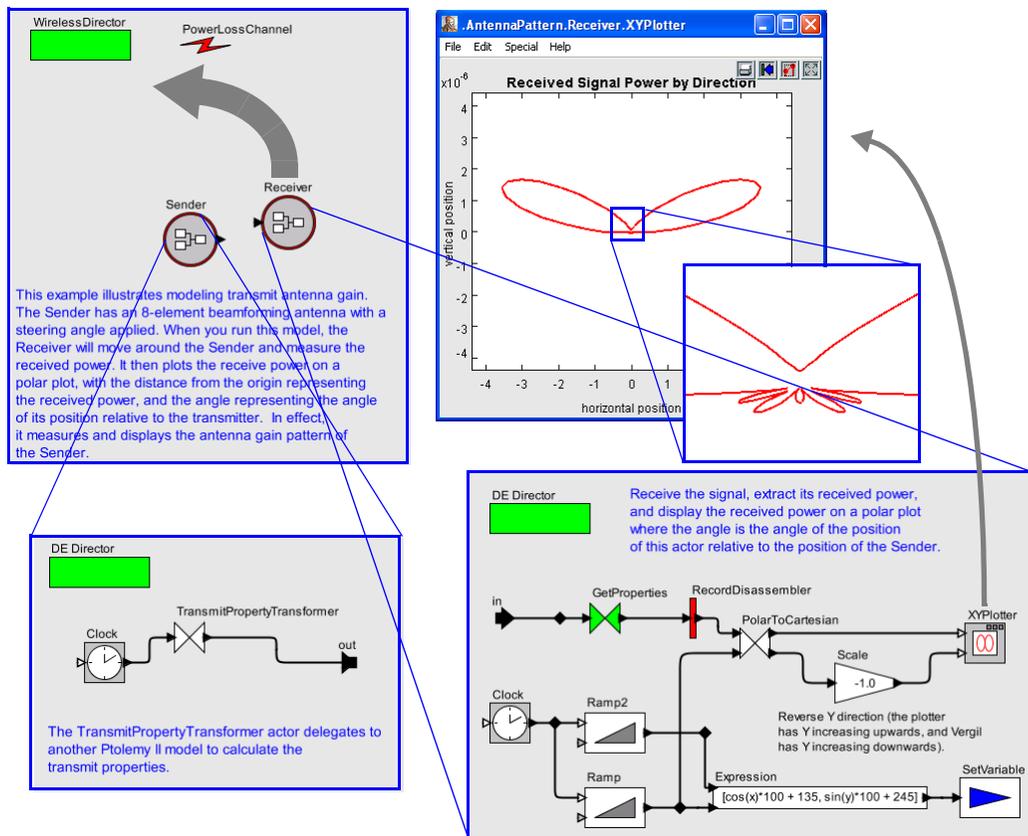


FIGURE 38. Model that includes a directional transmit antenna. As the model executes, the Receiver actor moves in a circular pattern around the transmitter and measures and plots the received power.

The antenna gain is calculated using the model shown in the middle of figure 39. This model uses two IterateOverArray actors (named “ArrayElements” and “Steering”) to model the antenna array elements and application of the steering vector. These actors are composite actors that execute their contained models once for each element of an input array. These actors are examples of *higher-order components*, and in this case enable the definition of a model where the number of antenna elements is given by a parameter rather than hardwired into the diagram. The same mechanism can be used to model the antenna gain pattern of the receiver.

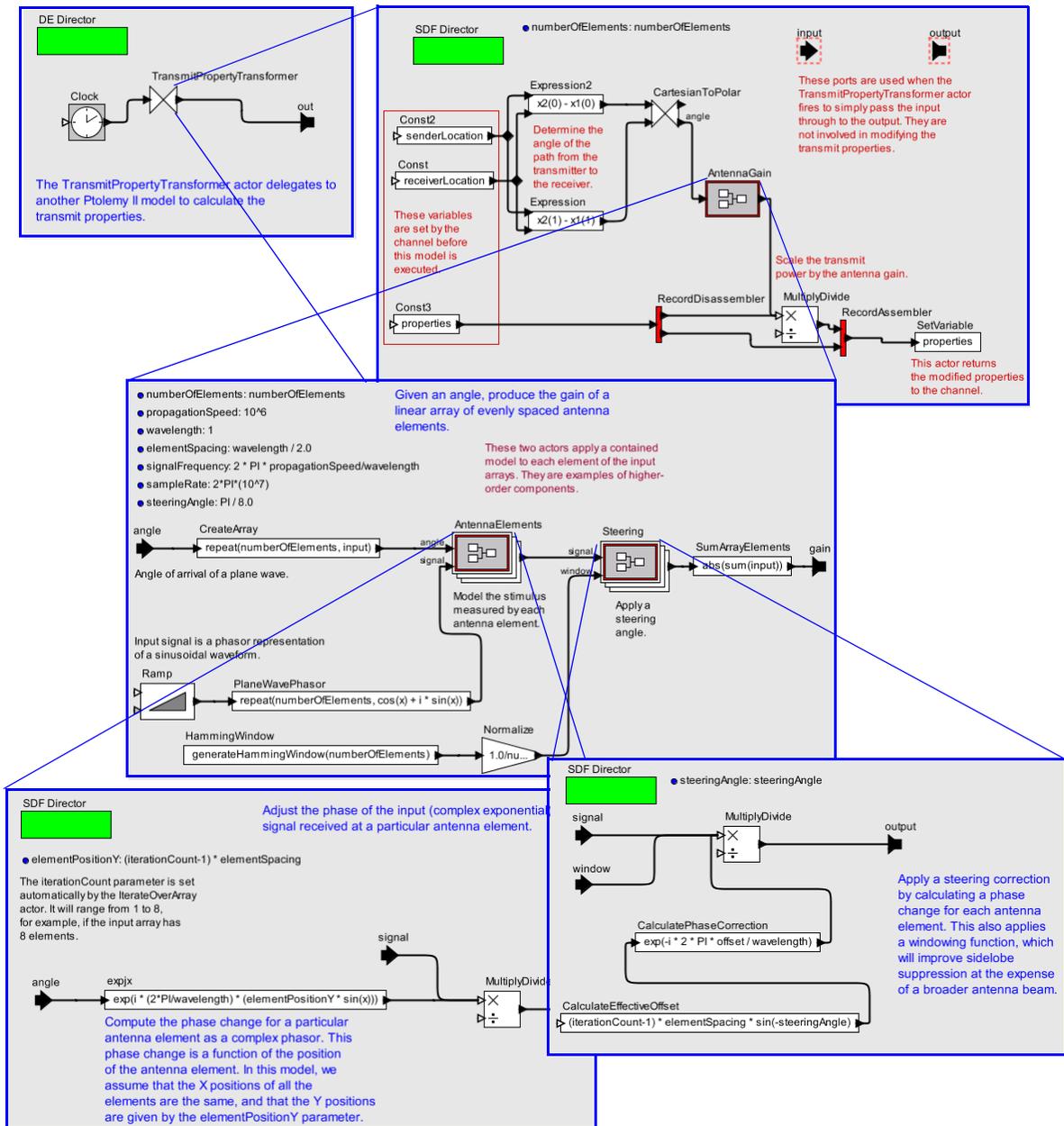


FIGURE 39. Transmitter design for the model in figure 38, showing how a Ptolemy II model (in this case a *synchronous dataflow* model) can be used to model transmission effects.

If there are multiple property transformers that are applicable to a particular transmission, then they are executed in an arbitrary order, so the operations they perform on the properties must be commutative. Typically, they select a field and multiply it by a constant.

4. Software Architecture

VisualSense is constructed by subclassing key classes in Ptolemy II. The extension to Ptolemy consists of a few new Java classes and some XML files. The classes are designed to be subclassed by model builders for customization, although non-trivial models can also be constructed without writing any Java code. In the latter case, sensor network nodes are specified using block diagrams and finite state machines.

The key classes in Ptolemy II (which define its meta model) are shown in figure 40. Executable components implement the Actor interface, and can be either atomic or composite. Atomic actors are defined in Java, while composite actors are assemblies of actors and relations. Each actor, whether atomic or not, contains ports, which are linked in a composite actor via relations. A top-level model is itself a composite actor, typically with no ports. Actors, ports and relations can all have attributes (parameters). One of the attributes is a director. The director plays a key role in Ptolemy II: it defines the semantics of a composite. It gives the concurrency model and the communication semantics. In VisualSense, the director implements the simulator. The WirelessDirector is an almost completely unmodified subclass of the pre-existing discrete-event director (DEDirector) in Ptolemy II.

The extensions that constitute VisualSense are shown in figure 40. A node in a wireless network is an actor that can be a subclass of either TypedAtomicActor or TypedCompositeActor. The difference between these is that for TypedAtomicActor, the behavior is defined in Java code, whereas for TypedCompositeActor, the behavior is defined by another Ptolemy II model, which is itself a composite of actors.

Actors that communicate wirelessly have ports that are instances of WirelessIOPort. As with any Ptolemy II port, the actor sends data by calling the *send* or *broadcast* method on the port. The *send* method permits specification of a numerically indexed subchannel, whereas the *broadcast* method will send to all subchannels.

In the case of WirelessIOPort, *send* and *broadcast* cannot determine the destination ports using block-diagram-style connectivity because there is no such connectivity. Instead, they identify an instance of WirelessChannel by name, and delegate to that instance to determine the destination(s) of the messages. The instance is specified by setting the *outsideChannel* parameter of the port equal to the name of the wireless channel (all actors at a given level of the hierarchy have unique names, a feature provided by the base class).

The WirelessChannel interface and the AtomicWirelessChannel base class, shown in figure 40, are designed for extensibility. They work together with WirelessIOPort, which uses the public method, *transmit*, to send data. That method takes three arguments, a *token*¹ to transmit, a source port, and a token representing transmit properties (transmit power, for example, as discussed below).

1. A token in Ptolemy II is a wrapper for data. Ptolemy II provides a rich set of data types encapsulated as tokens, including composite types such as arrays, matrices, and records (which have named fields). A sophisticated type system ensures validity of data that is exchanged via tokens. A rich expression language, described below, permits definition of tokens as expressions that can depend on parameters of actors or ports. Scoping rules limit the visibility of parameters according to the hierarchy of the model, thus avoiding the pitfalls of using global variables. For details, see [8].

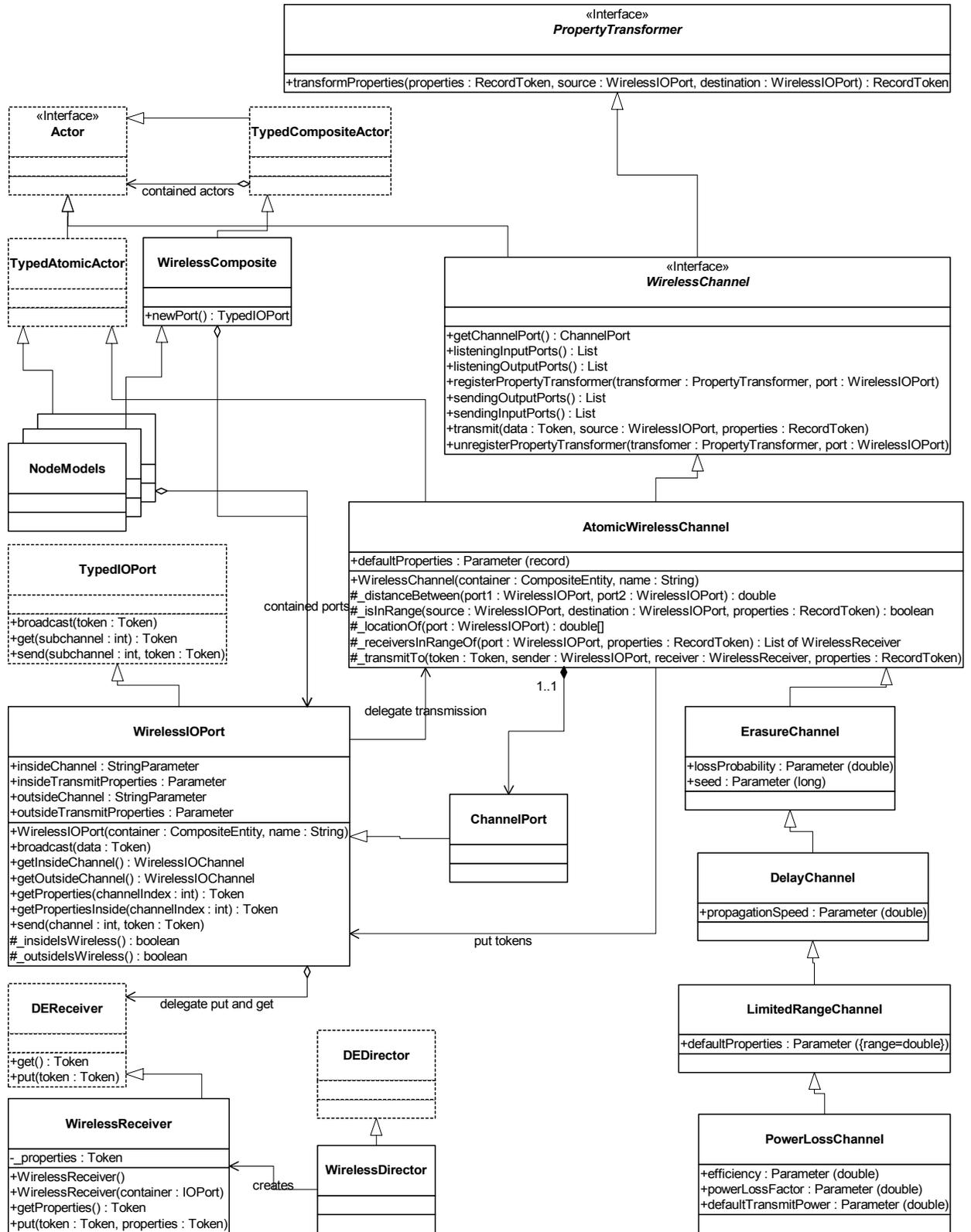


FIGURE 40. UML class diagram showing the key classes for wireless sensor network modeling. These classes plus some XML files specifying configuration information and libraries constitute VisualSense.

AtomicWirelessChannel has a suite of protected methods, indicated in the UML diagram by the leading pound sign (#); in the Ptolemy II coding style, protected methods have names that begin with leading underscores(_). These provide simple default behavior, but are intended to be overridden in subclasses to provide more sophisticated channel models. This is an example of the *strategy design pattern* [6], where the code providing the large-scale behavior delegates to protected methods for detailed behavior.

The default behavior of AtomicWirelessChannel is represented by the following pseudo code:

```
public void transmit(token, sender, properties) {
    foreach receiver in range {
        _transmitTo(token, sender, receiver, properties)
    }
}
```

To determine which receivers are in range, it calls the protected method `_receiversInRange()`, which by default returns all receivers contained by ports that refer to the same channel name as that specified by the sender. The `_transmitTo()` method by default uses the `public transformProperties()` method to modify the properties argument (see below) and then put the token and the modified properties into the receiver. The `transformProperties()` method applies any property transformers that are registered using the `registerPropertyTransformer()` method, but does nothing further. Thus, if there are no registered properties transformers, the default AtomicWirelessChannel has no range limitations and introduces no transmission degradations. We can now show through a series of examples how subclassing makes it easy to construct more detailed (and useful) channel models.

We illustrate the construction of model components such as channel models by subclassing with examples.

4.1 Erasure Channel

Consider a channel that randomly drops data. This can be defined as follows:

```
public class ErasureChannel extends AtomicWirelessChannel {
    ... specify constructor ...
    public Parameter lossProbability;
    public Parameter seed;
    private Random _random = new Random();
    public void transmit(token, sender, properties) {
        double experiment = _random.nextDouble();
        if (experiment >= lossProbability.doubleValue()) {
            super.transmit(token, sender, properties);
        }
    }
}
```

It is that simple. This channel adds to the base class a parameter called *lossProbability*. (The details of constructing the channel and this parameter are not shown, see [8]). The Java class `Random` is used to “throw the dice” to determine whether or not the transmission should actually occur.

Note that the above channel model might not be exactly what you want. In particular, it throws the dice once, and uses the result to decide whether or not to transmit to all recipient ports that are in range.

A better design might throw the dice once for each recipient port. We leave it as a (simple) exercise for the reader to see how to modify the above code to accomplish this.

4.2 Limited Range Channels

The above channels have unlimited range, in that any input port that references the channel by name is in range. This is because the default implementation of `_isInRange()` simply returns true. It is easy for a subclass to change this behavior. Consider, for example, a channel model that uses a distance threshold:

```
public class LimitedRangeChannel extends ErasureChannel {
    ... specify constructor ...
    public Parameter range;
    protected boolean _isInRange(
        source, destination, properties) {
        double distance = _distanceBetween(source, destination);
        if (distance <= range.doubleValue()) {
            return true;
        } else {
            return false;
        }
    }
}
```

This class overrides the `_isInRange()` method to simply check the distance between the source and the destination, returning true if the distance is below the specified *range* threshold. The `_isInRange()` method uses the `_locationOf()` method, which by default returns the (two-dimensional) location of the icon within the visual renditions of the model. Again, this yields a simplistic model, but nonetheless one that could be useful. It would be easy to build a variant where location is in three dimensional space and is specified by attributes attached to the sensor nodes. More sophisticated models of range rely on the *transmit properties* concept, which we explain next.

4.3 Transmit Properties

In the previous section, the range of wireless communication is a property of the channel. However, in many cases, it depends on properties of the transmitting sensor node and of extraneous features such as terrain. For example, a sensor node may have a power budget that depends on a battery model, and the power it uses for transmission will affect the range.

The argument called *properties* plays a central role. This argument is used to specify (model-dependent) information about a particular transmission. The *properties* argument is always a Record-Token, which is a composite data type in Ptolemy II that has named fields of arbitrary type. The `AtomicWirelessChannel` base class provides a *defaultProperties* parameter that defines the fields that are relevant for a particular channel.

A simple use of the *properties* field would be to specify the transmission range for a *particular* transmission. Indeed, the `LimitedRangeChannel` subclass of `AtomicWirelessChannel`, has a *defaultProperties* value of “{range = Infinity}”. A user of this channel could change this to, for example, “{range = 100.0}”, to represent that by default, transmissions have a range of 100 meters. An individ-

ual transmission can override this by setting the `outsideTransmitProperties` parameter of the sending port.

This model, however, is still simplistic. Communication ranges are typically not simple distances. More realistic models are supported by the `PowerLossChannel` subclass. This class has a parameter `powerPropagationFactor` whose default value is the expression “ $1.0 / (4 * \text{PI} * \text{distance} * \text{distance})$,” which assumes that the transmit power is uniformly distributed on a sphere of radius *distance*. The variable *distance* is a convenience variable provided in the scope in which this expression is evaluated. The user of this model may replace this expression with any expression using the rich Ptolemy II expression language, described below. The channel will then calculate the received power using the specified `powerPropagationFactor` and provide the received power to the receiving node via the `getProperties()` method of its input ports. The receiving node can then determine whether the signal has enough power to be received.

Much more sophisticated propagation models can be encapsulated and made available to the community as reusable components.

4.4 Antenna Gains and Terrain Models

Using the API as described so far, there appears to be no mechanism for implementing antenna gains or terrain models. These depend on the signal path from the transmitter to the receiver. However, close inspection reveals that the API is rich enough to accommodate these. In particular, the `WirelessChannel` interface has a key method, `registerPropertyTransformer()`, which can be used to register any object that implements the `TransformProperties` interface (which includes any object that implements `WirelessChannel`). An object that implements this interface is given the opportunity to modify the transmit properties of any transmission (or it can selectively indicate an interest only in transmissions coming from a particular port).

A transmit antenna model, for example, can be realized by an object that implements the `TransformProperties` interface. In fact, we can use Ptolemy II infrastructure to provide an object that uses another Ptolemy II model to implement the property transformation. Thus, the full suite of sophisticated signal processing capabilities of Ptolemy II are at the disposal of the builder of the antenna model.

The same goes for terrain models, although there is a caveat. Property transformers are required to implement modifications of the properties record that are commutative. That is, if there are several property transformers that can affect a particular transmission, the result of applying these transformers needs to be the same regardless of the order in which they are applied. For simple terrain models that apply only power loss, this will often be true. For some more sophisticated terrain models, however, it will not be true. Such models must be implemented as channels, subclassing for example the `PowerLossChannel`.

4.5 Delay Channels

The `DelayChannel` subclass of `EraseChannel` has a `propagationSpeed` parameter that the channel uses to determine the delay between transmission and reception of a signal. In the `DelayChannel` class, when the `transmit()` method is called, the channel calculates the delay to the specified location and requests that the director re-invoke it after that delay has elapsed (by calling the `fireAt()` method of the director, which places a request on the event queue). When it is reawakened, it delivers the message to the receiver.

5. Framework Infrastructure

The Ptolemy II framework provides some useful infrastructure.

5.1 Hierarchy and Heterogeneity

Ptolemy II supports hierarchical mixing of distinct models of computation. An inside model and its container model can have distinct directors. It is not uncommon for both directors to implement similar semantics. However, it is possible to have much bigger differences. In order to support this, the `WirelessIOPort` class in 40 can optionally specify both an *insideChannel* and an *outsideChannel*. If the outside channel is specified, then wireless communication is used on the outside. If the inside channel is specified, then wireless communication is used on the inside. Both can be used at the same time.

Another useful combination uses the continuous-time domain of Ptolemy II. This domain includes a `CTDirector` with a sophisticated numerical solver for ordinary differential equations and extensive support for hybrid systems modeling [2]. This can be used, for example, to construct sophisticated models of the physical mobility of mobile sensor platforms.

5.2 Type System

Ptolemy II includes a sophisticated type system [19]. In this type system, actors, parameters, and ports can all impose constraints on types, and a type resolution algorithm identifies the most specific types that satisfy all the constraints. By default, the type system in Ptolemy II includes a type constraint for each connection in a block diagram. However, in wireless models, these connections do not represent all the type constraints. In particular, every actor that sends data to a wireless channel requires that every recipient from that channel be able to accept that data type. `VisualSense` imposes this constraint in the `WirelessChannel` base class, so unless a particular model builder needs more sophisticated constraints, the model builder does not need to specify particular data types in the model. They will be inferred from the ultimate sources of the data and propagated throughout the model.

Note, however, that it would be unwise to explicitly model type constraints between every transmitter and every receiver using a channel. If there are n such users, this would be n^2 constraints, which for large n could bog down type resolution. As shown in 40, a channel contains a single port, an instance of `ChannelPort`. This is used to set up n type constraints, one to each user of the channel. This simplifies type resolution and keeps the static analysis of the model tractable even for large models.

5.3 Expressions

In `VisualSense`, models specify computations by composing actors. Many computations, however, are awkward to specify this way. A common situation is where we wish to evaluate a simple algebraic expression, such as “ $\sin(2\pi(x-1))$.” It is possible to express this computation by composing actors in a block diagram, but it is far more convenient to give it textually.

The expression language provides infrastructure for specifying algebraic expressions textually and for evaluating them. The expression language is used to specify the values of parameters, guards and actions in state machines, and for the calculation performed by the *Expression* actor. In fact, the expression language is part of the generic infrastructure in Ptolemy II, upon which `VisualSense` is built.

5.3.1 Expression Evaluator

`Vergil` provides an interactive *expression evaluator*, which is accessed through the File:New menu.

This operates like an interactive command shell, and is shown in figure 5.1. It supports a command history. To access the previously entered expression, type the up arrow or Control-P. To go back, type the down arrow or Control-N. The expression evaluator is useful for experimenting with expressions.

5.3.2 Simple Arithmetic Expressions

Constants and Literals. The simplest expression is a constant, which can be given either by the symbolic name of the constant, or by a literal. By default, the symbolic names of constants supported are PI, pi, E, e, true, false, i, j, NaN, Infinity, PositiveInfinity, NegativeInfinity, MaxUnsignedByte, MinUnsignedByte, MaxInt, MinInt, MaxLong, MinLong, MaxDouble, MinDouble. For example,

```
PI/2.0
```

is a valid expression that refers to the symbolic name “PI” and the literal “2.0.” The constants *i* and *j* are the imaginary number with value equal to the square root of -1 . The constant NaN is “not a number,” which for example is the result of dividing $0.0/0.0$. The constant Infinity is the result of dividing $1.0/0.0$. The constants that start with “Max” and “Min” are the maximum and minimum values for their corresponding types.

Numerical values without decimal points, such as “10” or “-3” are integers (type *int*). Numerical values with decimal points, such as “10.0” or “3.14159” are of type *double*. Numerical values without decimal points followed by the character “l” (el) or “L” are of type *long*. Unsigned integers followed by “ub” or “UB” are of type *unsignedByte*, as in “5ub”. An *unsignedByte* has a value between 0 and 255; note that it not quite the same as the Java byte, which has a value between -128 and 127.

Numbers of type *int*, *long*, or *unsignedByte* can be specified in decimal, octal, or hexadecimal. Numbers beginning with a leading “0” are octal numbers. Numbers beginning with a leading “0x” are hexadecimal numbers. For example, “012” and “0xA” are both equal to the integer 10.

A *complex* is defined by appending an “i” or a “j” to a double for the imaginary part. This gives a purely imaginary complex number which can then leverage the polymorphic operations in the Token

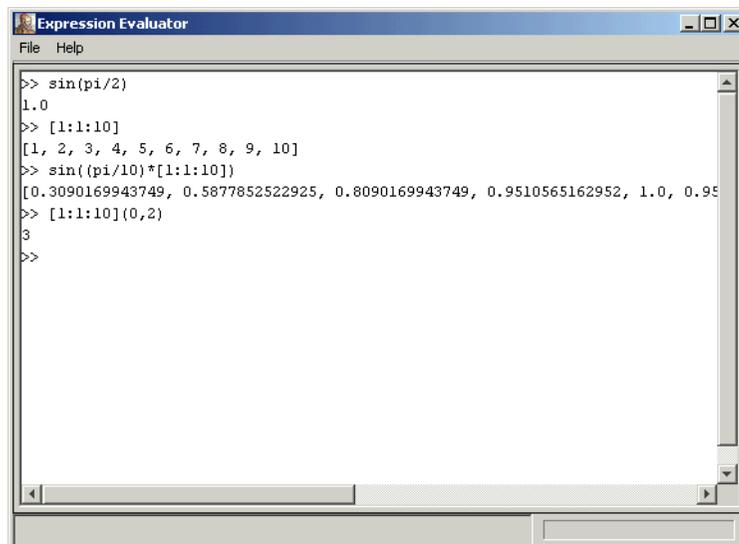


FIGURE 5.1. Expression evaluator, which is accessed through the File:New menu.

classes to create a general complex number. Thus “ $2 + 3i$ ” will result in the expected complex number. You can optionally write this “ $2 + 3*i$ ”.

Literal string constants are also supported. Anything between double quotes, “...”, is interpreted as a string constant. The following built-in string-valued constants are defined:

TABLE 3: String-valued constants defined in the expression language.

Variable name	Meaning	Property name	Example under Windows
PTII	The directory in which VisualSense is installed	ptolemy.ptII.dir	c:\tmp
HOME	The user home directory	user.home	c:\Documents and Settings\you
CWD	The current working directory	user.dir	c:\ptII

The value of these variables is the value of the Java virtual machine property, such as *user.home*. The properties *user.dir* and *user.home* are standard in Java. Their values are platform dependent; see the documentation for the `java.lang.System.getProperties()` method for details. Note that *user.dir* and *user.home* are usually not readable in unsigned applets, in which case, attempts to use these variables in an expression will result in an exception. Vergil will display all the Java properties if you invoke JVM Properties in the View menu of a Graph Editor.

The *ptolemy.ptII.dir* property is set automatically when VisualSense is started up. The `constants()` utility function returns a record with all the globally defined constants. If you open the expression evaluator and invoke this function, you will see that its value is something like:

```
{CWD="C:\ptII\ptolemy\data\expr", E=2.718281828459,
HOME="C:\Documents and Settings\eal", Infinity=Infinity, MaxDouble=1.7976931348623E308,
MaxInt=2147483647, MaxLong=9223372036854775807L, MaxUnsignedByte=255ub,
MinDouble=4.9E-324, MinInt=-2147483648, MinLong=-9223372036854775808L,
MinUnsignedByte=0ub, NaN=NaN, NegativeInfinity=-Infinity, PI=3.1415926535898,
PTII="c:\ptII", PositiveInfinity=Infinity, boolean=false, complex=0.0 + 0.0i,
double=0.0, e=2.718281828459, false=false, fixedpoint=fix(0.0,2,1),
general=present, i=0.0 + 1.0i, int=0, j=0.0 + 1.0i, long=0L,
matrix=[], object=object(null), pi=3.1415926535898, scalar=present, string="",
true=true, unknown=present, unsignedByte=0ub}
```

Variables. Expressions can contain identifiers that are references to variables within the *scope* of the expression. For example,

```
PI*x/2.0
```

is valid if “x” is a variable in scope. In the expression evaluator, the variables that are in scope include the built-in constants plus any assignments that have been previously made. For example,

```
>> x = pi/2
1.5707963267949
>> sin(x)
1.0
>>
```

In the context of VisualSense models, the variables in scope include all parameters defined at the same level of the hierarchy or higher. So for example, if an actor has a parameter named “x” with value 1.0, then another parameter of the same actor can have an expression with value “PI*x/2.0”, which will evaluate to $\pi/2$.

Consider a parameter P in actor X which is in turn contained by composite actor Y . The scope of an expression for P includes all the parameters contained by X and Y , plus those of the container of Y , its container, etc. That is, the scope includes any parameters defined above in the hierarchy.

You can add parameters to actors (composite or not) by right clicking on the actor, selecting “Configure” and then clicking on “Add”, or by dragging in a parameter from the *utilities* library. Thus, you can add variables to any scope, a capability that serves the same role as the “let” construct in many functional programming languages.

Operators. The arithmetic operators are +, −, *, /, ^, and %. Most of these operators operate on most data types, including arrays, records, and matrices. The ^ operator computes “to the power of” or exponentiation where the exponent can only be an *int* or an *unsignedByte*.

The *unsignedByte*, *int* and *long* types can only represent integer numbers. Operations on these types are integer operations, which can sometimes lead to unexpected results. For instance, 1/2 yields 0 if 1 and 2 are integers, whereas 1.0/2.0 yields 0.5. The exponentiation operator ‘^’ when used with negative exponents can similarly yield unexpected results. For example, 2^{-1} is 0 because the result is computed as $1/(2^1)$.

The `%` operation is a *modulo* or *remainder* operation. The result is the remainder after division. The sign of the result is the same as that of the dividend (the left argument). For example,

```
>> 3.0 % 2.0
1.0
>> -3.0 % 2.0
-1.0
>> -3.0 % -2.0
-1.0
>> 3.0 % -2.0
1.0
```

The magnitude of the result is always less than the magnitude of the divisor (the right argument). Note that when this operator is used on doubles, the result is not the same as that produced by the `remainder()` function (see Table 6 on page 60). For instance,

```
>> remainder(-3.0, 2.0)
1.0
```

The `remainder()` function calculates the IEEE 754 standard remainder operation. It uses a rounding division rather than a truncating division, and hence the sign can be positive or negative, depending on complicated rules (see page 56). For example, counterintuitively,

```
>> remainder(3.0, 2.0)
-1.0
```

When an operator involves two distinct types, the expression language has to make a decision about which type to use to implement the operation. If one of the two types can be converted without loss into the other, then it will be. For instance, *int* can be converted losslessly to *double*, so `1.0/2` will result in 2 being first converted to 2.0, so the result will be 0.5. Among the scalar types, *unsignedByte* can be converted to anything else, *int* can be converted to *double*, and *double* can be converted to *complex*. Note that *long* cannot be converted to *double* without loss, nor vice versa, so an expression like `2.0/2L` yields the following error message:

```
Error evaluating expression "2.0/2L"
  in .Expression.evaluator
Because:
divide method not supported between ptolemy.data.DoubleToken '2.0' and
ptolemy.data.LongToken '2L' because the types are incomparable.
```

All scalar types have limited precision and magnitude. As a result of this, arithmetic operations are subject to underflow and overflow.

- For *double* numbers, overflow results in the corresponding positive or negative infinity. Underflow (i.e. the precision does not suffice to represent the result) will yield zero.
- For integer types and *fixedpoint*, overflow results in wraparound. For instance, while the value of `MaxInt` is 2147483647, the expression `MaxInt + 1` yields `-2147483648`. Similarly, while `MaxUnsignedByte` has value 255, `MaxUnsignedByte + 1` has value 0. Note, however, that

`MaxUnsignedByte + 1` yields 256, which is an *int*, not an *unsignedByte*. This is because `MaxUnsignedByte` can be losslessly converted to an *int*, so the addition is *int* addition, not *unsignedByte* addition.

The bitwise operators are `&`, `|`, `#`, and `~`. They operate on *boolean*, *unsignedByte*, *int* and *long* (but not *fixedpoint*, *double* or *complex*). The operator `&` is bitwise AND, `~` is bitwise NOT, and `|` is bitwise OR, and `#` is bitwise XOR (exclusive or, after MATLAB).

The relational operators are `<`, `<=`, `>`, `>=`, `==` and `!=`. They return type *boolean*. Note that these relational operators check the values when possible, irrespective of type. So, for example,

```
1 == 1.0
```

returns *true*. If you wish to check for equality of both type and value, use the `equals()` method, as in

```
>> 1.equals(1.0)
false
```

Boolean-valued expressions can be used to give conditional values. The syntax for this is

```
boolean ? value1 : value2
```

If the boolean is true, the value of the expression is `value1`; otherwise, it is `value2`.

The logical boolean operators are `&&`, `||`, `!`, `&` and `|`. They operate on type *boolean* and return type *boolean*. The difference between logical `&&` and logical `&` is that `&` evaluates all the operands regardless of whether their value is now irrelevant. Similarly for logical `||` and `|`. This approach is borrowed from Java. Thus, for example, the expression “`false && x`” will evaluate to *false* irrespective of whether `x` is defined. On the other hand, “`false & x`” will throw an exception.

The `<<` and `>>` operators performs arithmetic left and right shifts respectively. The `>>>` operator performs a logical right shift, which does not preserve the sign. They operate on *unsignedByte*, *int*, and *long*.

Comments. In expressions, anything inside `/* . . . */` is ignored, so you can insert comments.

5.3.3 Uses of Expressions

Parameters. The values of most parameters of actors can be given as expressions¹. The variables in the expression refer to other parameters that are in scope, which are those contained by the same container or some container above in the hierarchy. They can also reference variables in a *scope-extending attribute*, which includes variables defining units. Adding parameters to actors is straightforward, as explained in the previous chapter.

String Parameters. Some parameters have values that are always strings of characters. Such parameters support a simple string substitution mechanism where the value of the string can reference other

1. The exceptions are parameters that are strictly string parameters, in which case the value of the parameter is the literal string, not the string interpreted as an expression, as for example the *function* parameter of the *TrigFunction* actor, which can take on only “sin,” “cos,” “tan,” “asin,” “acos,” and “atan” as values.

parameters in scope by name using the syntax $\$name$, where $name$ is the name of the parameter in scope. For example, the StringCompare actor in figure 5.2 has as the value of *firstString* “The answer is \$PI”. This references the built-in constant PI. The value of *secondString* is “The answer is 3.1415926535898”. As shown in the figure, these two strings are deemed to be equal because \$PI is replaced with the value of PI.

Port Parameters. It is possible to define a parameter that is also a port. Such a *PortParameter* provides a default value, which is specified like the value of any other parameter. When the corresponding port receives data, however, the default value is overridden with the value provided at the port. Thus, this object functions like a parameter and a port. The current value of the PortParameter is accessed like that of any other parameter. Its current value will be either the default or the value most recently received on the port.

A PortParameter might be contained by an atomic actor or a composite actor. To put one in a composite actor, drag it into a model from the *utilities* library, as shown in figure 5.3. The resulting icon is actually a combination of two icons, one representing the port, and the other representing the parameter. These can be moved separately, but doing so might create confusion, so we recommend selecting both by clicking and dragging over the pair and moving both together.

To be useful, a PortParameter has to be given a name (the default name, “portParameter,” is not very compelling). To change the name, right click on the icon and select “Customize Name,” as shown in figure 5.3. In the figure, the name is set to “noiseLevel.” Then set the default value by either double clicking or selecting “Configure.” In the figure, the default value is set to 10.0.

An example of a library actor that uses a PortParameter is the Sinewave actor, which is found in the *sources* library in Vergil. It is shown in figure 5.4. If you double click on this actor, you can set the default values for *frequency* and *phase*. But both of these values can also be set by the corresponding ports, which are shown with grey fill.

Expression Actor. The *Expression* actor is a particularly useful actor found in the *math* library. By default, it has one output and no inputs, as shown in Figure 5.5(a). The first step in using it is to add ports, as shown in (b) and (c), resulting in a new icon as shown in (d). Note: In (c) when you click on Add, you will be prompted for a Name (pick one) and a Class. Leave the Class entry blank and click OK. You then specify an expression using the port names, as shown in (e), resulting in the icon shown

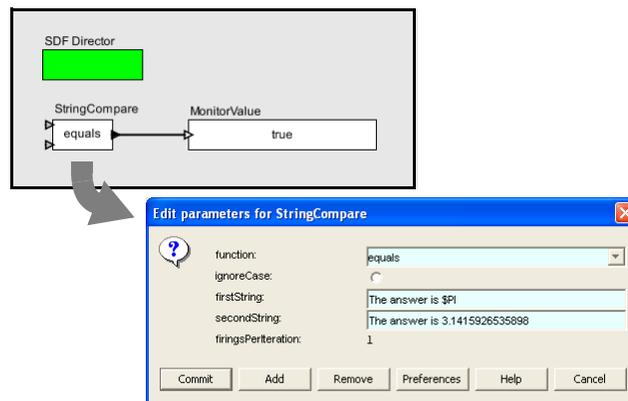


FIGURE 5.2. String parameters are indicated in the parameter editor boxes by a light blue background. A string parameter can include references to variables in scope with $\$name$, where $name$ is the name of the variable. In this example, the built-in constant PI is referenced by name in the first

in (f).

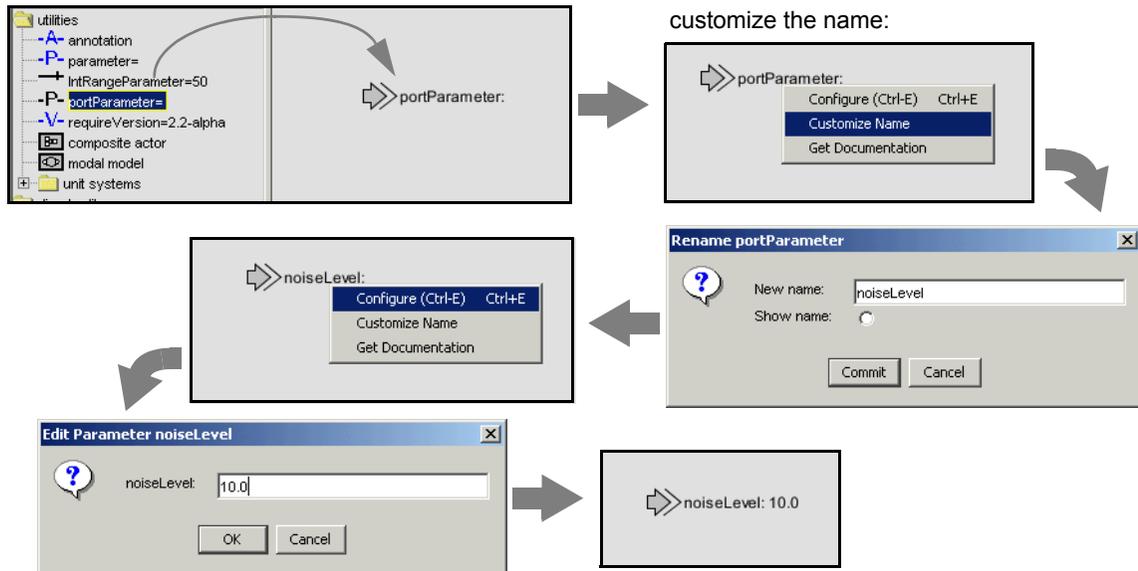


FIGURE 5.3. A *portParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful, and set its default value.

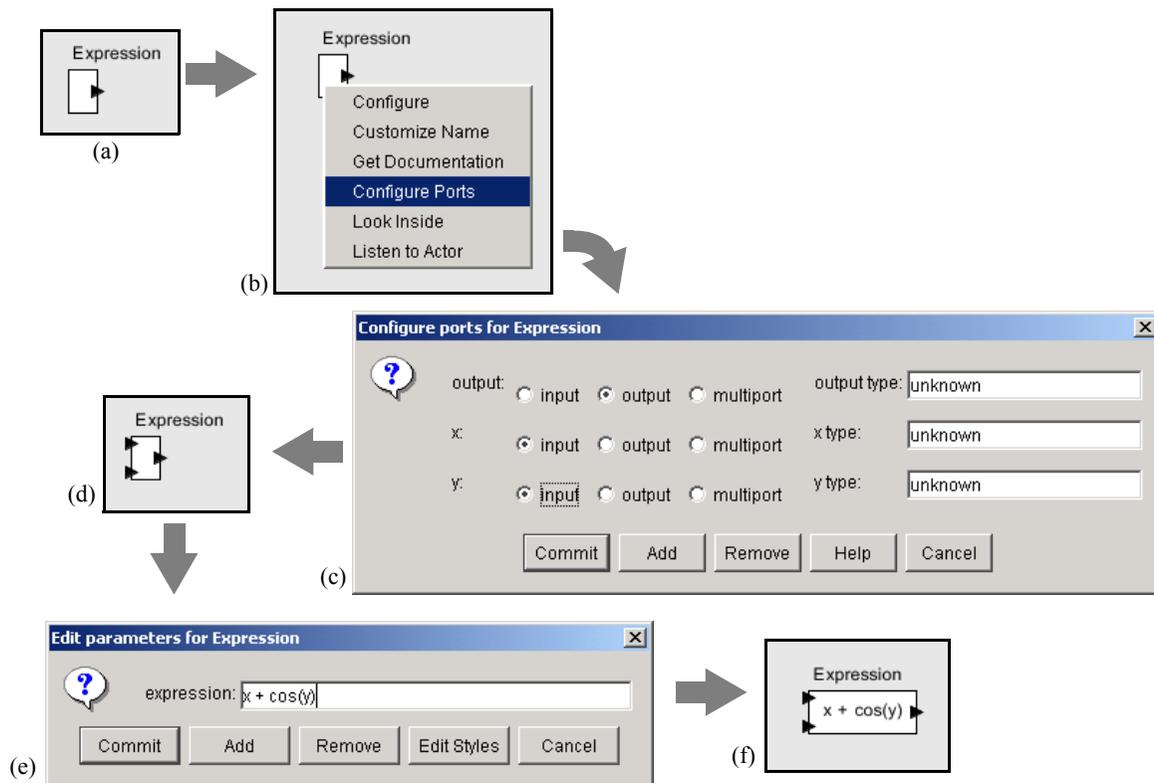


FIGURE 5.5. Illustration of the *Expression* actor.

State Machines. Expressions give the guards for state transitions, as well as the values used in actions that produce outputs and actions that set values of parameters in the refinements of destination states. This mechanism was explained in the previous chapter.

5.4 Composite Data Types

5.4.1 Arrays

Arrays are specified with curly brackets, e.g., “{1, 2, 3}” is an array of *int*, while “{“x”, “y”, “z”}” is an array of *string*. The types are denoted “{int}” and “{string}” respectively. An array is an ordered list of tokens of any type, with the only constraint being that the elements all have

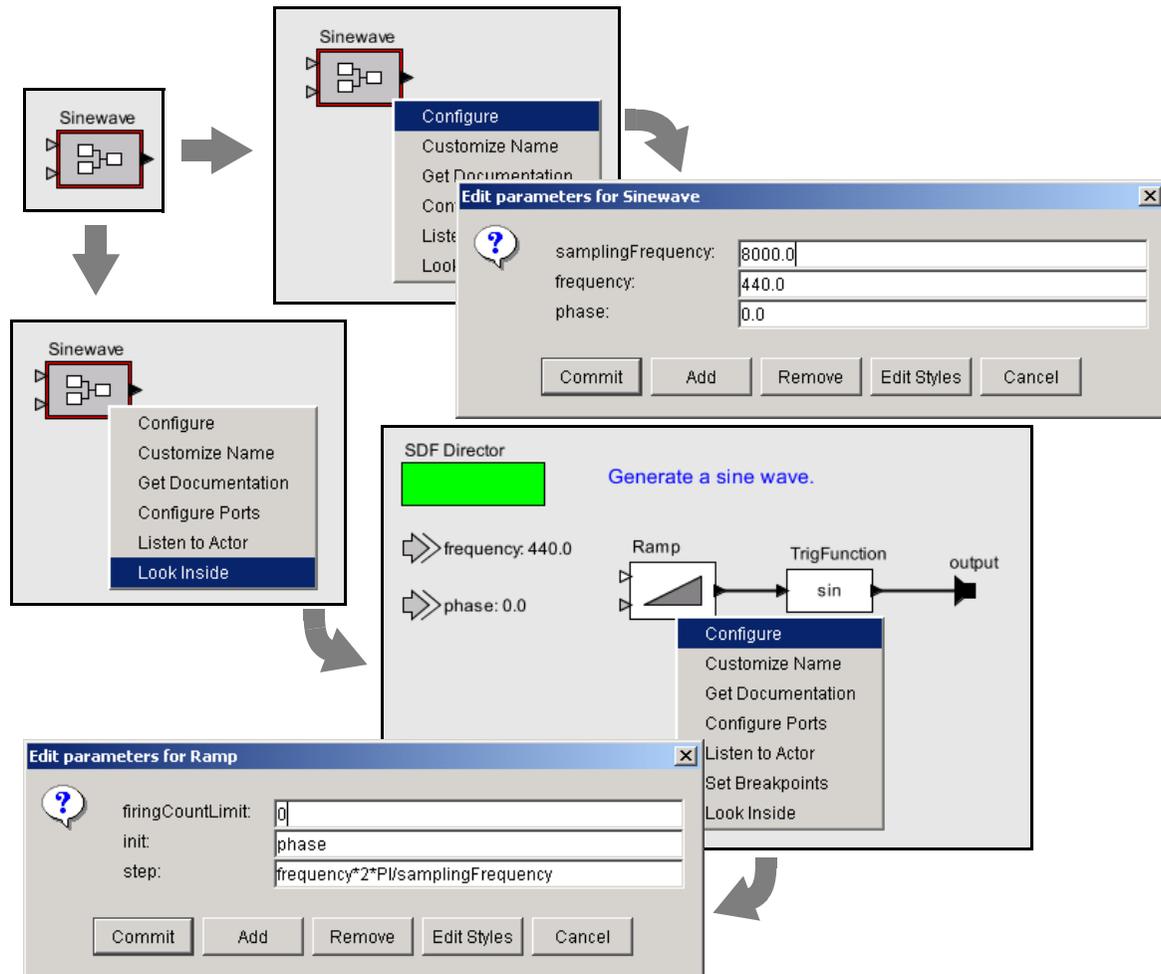


FIGURE 5.4. Sinewave actor, showing its port parameters, and their use at the lower level of the hierarchy.

the same type. If an array is given with mixed types, the expression evaluator will attempt to losslessly convert the elements to a common type. Thus, for example,

```
{1, 2.3}
```

has value with type `{double}`:

```
{1.0, 2.3}
```

The elements of the array can be given by expressions, as in the example “`{2*pi, 3*pi}`.” Arrays can be nested; for example, “`{{1, 2}, {3, 4, 5}}`” is an array of arrays of integers. The elements of an array can be accessed as follows:

```
>> {1.0, 2.3}(1)
2.3
```

which yields 2.3. Note that indexing begins at 0. Of course, if *name* is the name of a variable in scope whose value is an array, then its elements may be accessed similarly, as shown in this example:

```
>> x = {1.0, 2.3}
{1.0, 2.3}
>> x(0)
1.0
```

Arithmetic operations on arrays are carried out element-by-element, as shown by the following examples:

```
>> {1, 2}*{2, 2}
{2, 4}
>> {1, 2}+{2, 2}
{3, 4}
>> {1, 2}-{2, 2}
{-1, 0}
>> {1, 2}^2
{1, 4}
>> {1, 2}%{2, 2}
{1, 0}
```

An array can be checked for equality with another array as follows:

```
>> {1, 2}=={2, 2}
false
>> {1, 2}!={2, 2}
true
```

For other comparisons of arrays, use the `compare()` function (see Table 6 on page 60). As with scalars, testing for equality using the `==` or `!=` operators tests the values, independent of type. For example,

```
>> {1, 2}=={1.0, 2.0}
true
```

5.4.2 Matrices

In VisualSense, *arrays* are ordered sets of tokens. VisualSense also supports *matrices*, which are more specialized than arrays. They contain only certain primitive types, currently *boolean*, *complex*, *double*, *fixedpoint*, *int*, and *long*. Currently *unsignedByte* matrices are not supported. Matrices cannot contain arbitrary tokens, so they cannot, for example, contain matrices. They are intended for data intensive computations.

Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows. E.g., “[1, 2, 3; 4, 5, 5+1]” gives a two by three integer matrix (2 rows and 3 columns). Note that an array or matrix element can be given by an expression. A row vector can be given as “[1, 2, 3]” and a column vector as “[1; 2; 3]”. Some MATLAB-style array constructors are supported. For example, “[1:2:9]” gives an array of odd numbers from 1 to 9, and is equivalent to “[1, 3, 5, 7, 9].” Similarly, “[1:2:9; 2:2:10]” is equivalent to “[1, 3, 5, 7, 9; 2, 4, 6, 8, 10].” In the syntax “[*p*:*q*:*r*]”, *p* is the first element, *q* is the step between elements, and *r* is an upper bound on the last element. That is, the matrix will not contain an element larger than *r*. If a matrix with mixed types is specified, then the elements will be converted to a common type, if possible. Thus, for example, “[1.0, 1]” is equivalent to “[1.0, 1.0],” but “[1.0, 1L]” is illegal (because there is no common type to which both elements can be converted losslessly).

Reference to elements of matrices have the form “*matrix*(*n*, *m*)” or “*name*(*n*, *m*)” where *name* is the name of a matrix variable in scope, *n* is the row index, and *m* is the column index. Index numbers start with zero, as in Java, not 1, as in MATLAB. For example,

```
>> [1, 2; 3, 4](0,0)
1
>> a = [1, 2; 3, 4]
[1, 2; 3, 4]
>> a(1,1)
4
```

Matrix multiplication works as expected. For example, as seen in the expression evaluator (see figure 5.1),

```
>> [1, 2; 3, 4]*[2, 2; 2, 2]
[6, 6; 14, 14]
```

Of course, if the dimensions of the matrix don't match, then you will get an error message. To do elementwise multiplication, use the `multiplyElements()` function (see Table 7 on page 62). Matrix addition and subtraction are elementwise, as expected, but the division operator is not supported. Elementwise division can be accomplished with the `divideElements()` function, and multiplication by a matrix inverse can be accomplished using the `inverse()` function (see Table 7 on page 62). A matrix can be

raised to an *int* or *unsignedByte* power, which is equivalent to multiplying it by itself some number of times. For instance,

```
>> [3, 0; 0, 3]^3
[27, 0; 0, 27]
```

A matrix can also be multiplied or divided by a scalar, as follows:

```
>> [3, 0; 0, 3]*3
[9, 0; 0, 9]
```

A matrix can be added to a scalar. It can also be subtracted from a scalar, or have a scalar subtracted from it. For instance,

```
>> 1-[3, 0; 0, 3]
[-2, 1; 1, -2]
```

A matrix can be checked for equality with another matrix as follows:

```
>> [3, 0; 0, 3]!=[3, 0; 0, 6]
true
>> [3, 0; 0, 3]==[3, 0; 0, 3]
true
```

For other comparisons of matrices, use the `compare()` function (see Table 6 on page 60). As with scalars, testing for equality using the `==` or `!=` operators tests the values, independent of type. For example,

```
>> [1, 2]==[1.0, 2.0]
true
```

To get type-specific equality tests, use the `equals()` method, as in the following examples:

```
>> [1, 2].equals([1.0, 2.0])
false
>> [1.0, 2.0].equals([1.0, 2.0])
true
>>
```

5.4.3 Records

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

Fields may be accessed using the period operator. For example,

```
{a=1,b=2}.a
```

yields 1. You can optionally write this as if it were a method call:

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

The arithmetic operators `+`, `-`, `*`, `/`, and `%` can be applied to records. If the records do not have identical fields, then the operator is applied only to the fields that match, and the result contains only the fields that match. Thus, for example,

```
{foodCost=40, hotelCost=100} + {foodCost=20, taxiCost=20}
```

yields the result

```
{foodCost=60}
```

You can think of an operation as a set intersection, where the operation specifies how to merge the values of the intersecting fields. You can also form an intersection without applying an operation. In this case, using the `intersect()` function, you form a record that has only the common fields of two specified records, with the values taken from the first record. For example,

```
>> intersect({a=1, c=2}, {a=3, b=4})  
{a=1}
```

Records can be joined (think of a set union) without any operation being applied by using the `merge()` function. This function takes two arguments, both of which are record tokens. If the two record tokens have common fields, then the field value from the first record is used. For example,

```
merge({a=1, b=2}, {a=3, c=3})
```

yields the result `{a=1, b=2, c=3}`.

Records can be compared, as in the following examples:

```
>> {a=1, b=2}!={a=1, b=2}
false
>> {a=1, b=2}!={a=1, c=2}
true
```

Note that two records are equal only if they have the same field labels and the values match. As with scalars, the values match irrespective of type. For example:

```
>> {a=1, b=2}=={a=1.0, b=2.0+0.0i}
true
```

The order of the fields is irrelevant. Hence

```
>> {a=1, b=2}=={b=2, a=1}
true
```

Moreover, record fields are reported in alphabetical order, irrespective of the order in which they are defined. For example,

```
>> {b=2, a=1}
{a=1, b=2}
```

To get type-specific equality tests, use the `equals()` method, as in the following examples:

```
>> {a=1, b=2}.equals({a=1.0, b=2.0+0.0i})
false
>> {a=1, b=2}.equals({b=2, a=1})
true
>>
```

5.5 Invoking Methods in Expressions

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

```
{1, 2, 3}.length()
{a=1, b=2, c=3}.length()
```

each of which returns the integer 3.

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

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

which returns 3, and

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

which returns 2, and

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

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

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

5.6 Defining Functions in Expressions

The expression language supports definition of functions. The syntax is:

```
function(arg1:Type, arg2:Type...)  
    function body
```

where “function” is the keyword for defining a function. The type of an argument can be left unspecified, in which case the expression language will attempt to infer it. The function body gives an expression that defines the return value of the function. The return type is always inferred based on the argument type and the expression. For example:

```
function(x:double) x*5.0
```

defines a function that takes a *double* argument, multiplies it by 5.0, and returns a double. The return value of the above expression is the function itself. Thus, for example, the expression evaluator yields:

```
>> function(x:double) x*5.0  
(function(x:double) (x*5.0))  
>>
```

To apply the function to an argument, simply do

```
>> (function(x:double) x*5.0) (10.0)  
50.0  
>>
```

Alternatively, in the expression evaluator, you can assign the function to a variable, and then use the variable name to apply the function. For example,

```
>> f = function(x:double) x*5.0
(function(x:double) (x*5.0))
>> f(10)
50.0
>>
```

Functions can be passed as arguments to certain “higher-order functions” that have been defined (see table Table 10 on page 66). For example, the `iterate()` function takes three arguments, a function, an integer, and an initial value to which to apply the function. It applies the function first to the initial value, then to the result of the application, then to that result, collecting the results into an array whose length is given by the second argument. For example, to get an array whose values are multiples of 3, try

```
>> iterate(function(x:int) x+3, 5, 0)
{0, 3, 6, 9, 12}
```

The function given as an argument simply adds three to its argument. The result is the specified initial value (0) followed by the result of applying the function once to that initial value, then twice, then three times, etc.

Another useful higher-order function is the `map()` function. This one takes a function and an array as arguments, and simply applies the function to each element of the array to construct a result array. For example,

```
>> map(function(x:int) x+3, {0, 2, 3})
{3, 5, 6}
```

A typical use of functions in a VisualSense model is to define a parameter in a model whose value is a function. Suppose that the parameter named “f” has value “`function(x:double) x*5.0`”. Then within the scope of that parameter, the expression “`f(10.0)`” will yield result 50.0.

Functions can also be passed along connections in a VisualSense model. Consider the model shown in figure 5.6. In that example, the Const actor defines a function that simply squares the argu-

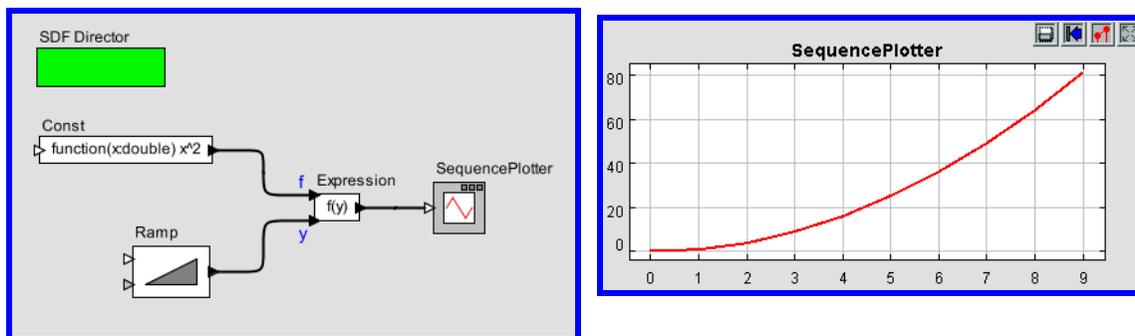


FIGURE 5.6. Example of a function being passed from one actor to another.

ment. Its output, therefore, is a token with type *function*. That token is fed to the “f” input of the Expression actor. The expression uses this function by applying it to the token provided on the “y” input. That token, in turn, is supplied by the Ramp actor, so the result is the curve shown in the plot on the right.

A more elaborate use is shown in figure 5.7. In that example, the Const actor produces a function, which is then used by the Expression actor to create new function, which is then used by Expression2 to perform a calculation. The calculation performed here adds the output of the Ramp to the square of the output of the Ramp.

Functions can be recursive, as illustrated by the following (rather arcane) example:

```
>> fact = function(x:int,f:(function(x,f) int)) (x<1?1:x*f(x-1,f))
(function(x:int, f:function(a0:general, a1:general) int)
(x<1)?1:(x*f((x-1), f)))
>> factorial = function(x:int) fact(x,fact)
(function(x:int) (function(x:int, f:function(a0:general, a1:general)
int) (x<1)?1:(x*f((x-1), f)))(x, (function(x:int, f:function(a0:gen-
eral, a1:general) int) (x<1)?1:(x*f((x-1), f)))))
>> map(factorial, [1:1:5].toArray())
{1, 2, 6, 24, 120}
>>
```

The first expression defines a function named “fact” that takes a function as an argument, and if the argument is greater than or equal to 1, uses that function recursively. The second expression defines a new function “factorial” using “fact.” The final command applies the factorial function to an array to compute factorials.

5.7 Built-In Functions

The expression language includes a set of functions, such as sin(), cos(), etc. The functions currently available are shown in the tables in the appendix, which also show the argument types and return

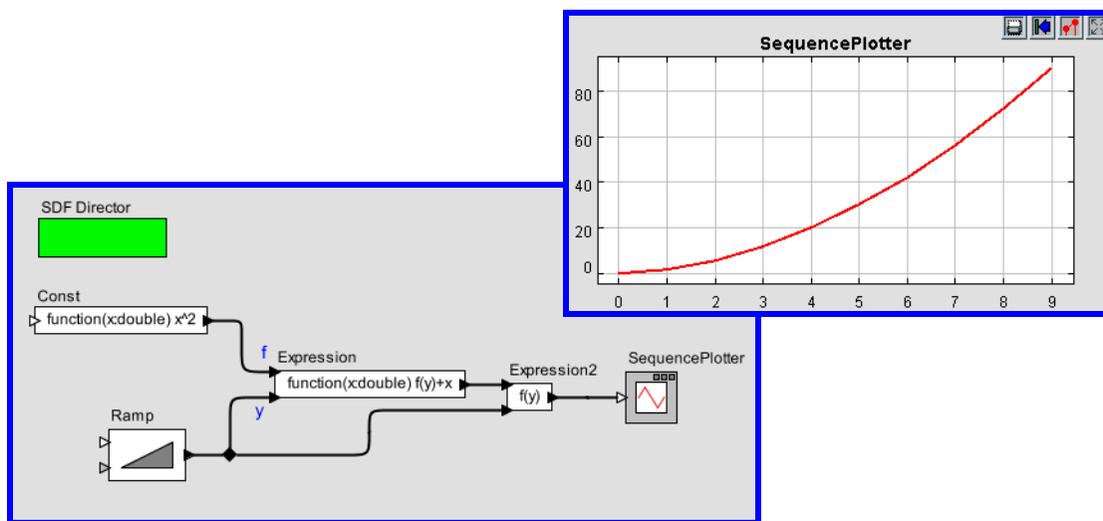


FIGURE 5.7. More elaborate example with functions passed between actors.

types.

In most cases, a function that operates on scalar arguments can also operate on arrays and matrices. Thus, for example, you can fill a row vector with a sine wave using an expression like

```
sin([0.0:PI/100:1.0])
```

Or you can construct an array as follows,

```
sin({0.0, 0.1, 0.2, 0.3})
```

Functions that operate on type *double* will also generally operate on *int* or *unsignedByte*, because these can be losslessly converted to *double*, but not generally on *long* or *complex*.

Tables of available functions are shown in the appendix. For example, Table 5 on page 59 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be *double*. For example,

```
>> cos(0)
1.0
```

These functions will also operate on matrices and arrays, in addition to the scalar types shown in the table, as illustrated above. The result will be a matrix or array of the same size as the argument, but always containing elements of type *double*

Table 6 on page 60 shows other arithmetic functions beyond the trigonometric functions. As with the trigonometric functions, those that indicate that they operate on *double* will also work on *int* and *unsignedByte*, and unless they indicate otherwise, they will return whatever they return when the argument is *double*. Those functions in the table that take scalar arguments will also operate on matrices and arrays. For example, since the table indicates that the `max()` function can take *int*, *int* as arguments, then by implication, it can also take *{int}*, *{int}*. For example,

```
>> max({1, 2}, {2, 1})
{2, 2}
```

Notice that the table also indicates that `max()` can take *{int}* as an argument. E.g.

```
>> max({1, 2, 3})
3
```

In the former case, the function is applied pointwise to the two arguments. In the latter case, the returned value is the maximum over all the contents of the single argument.

Table 7 shows functions that only work with matrices, arrays, or records (that is, there is no corresponding scalar operation). Recall that most functions that operate on scalars will also operate on arrays and matrices. Table 8 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the `eval()` function is the most flexible (see page 55).

A few of the functions have sufficiently subtle properties that they require further explanation. That explanation is here.

eval() and traceEvaluation()

The built-in function `eval()` will evaluate a string as an expression in the expression language. For example,

```
eval("[1.0, 2.0; 3.0, 4.0]")
```

will return a matrix of doubles. The following combination can be used to read parameters from a file:

```
eval(readFile("filename"))
```

where the *filename* can be relative to the current working directory (where VisualSense was started, as reported by the property `user.dir`), the user's home directory (as reported by the property `user.home`), or the classpath, which includes the directory tree in which VisualSense is installed.

Note that if `eval()` is used in an Expression actor, then it will be impossible for the type system to infer any more specific output type than *general*. If you need the output type to be more specific, then you will need to cast the result of `eval()`. For example, to force it to type *double*:

```
>> cast(double, eval("pi/2"))  
1.5707963267949
```

The `traceEvaluation()` function evaluates an expression given as a string, much like `eval()`, but instead of reporting the result, reports exactly how the expression was evaluated. This can be used to debug expressions, particularly when the expression language is extended by users.

random(), gaussian()

The functions `random()` and `gaussian()` shown in Table 6 on page 60 return one or more random numbers. With the minimum number of arguments (zero or two, respectively), they return a single number. With one additional argument, they return an array of the specified length. With a second additional argument, they return a matrix with the specified number of rows and columns.

There is a key subtlety when using these functions in VisualSense. In particular, they are evaluated only when the expression within which they appear is evaluated. The result of the expression may be used repeatedly without re-evaluating the expression. Thus, for example, if the *value* parameter of the *Const* actor is set to "`random()`", then its output will be a random constant, i.e., it will not change on each firing. The output will change, however, on successive runs of the model. In contrast, if this is used in an Expression actor, then each firing triggers an evaluation of the expression, and consequently will result in a new random number.

property()

The `property()` function accesses system properties by name. Some possibly useful system properties are:

- `ptolemy.ptII.dir`: The directory in which VisualSense is installed.
- `ptolemy.ptII.dirAsURL`: The directory in which VisualSense is installed, but represented as a URL.
- `user.dir`: The current working directory, which is usually the directory in which the current executable was started.

remainder()

This function computes the remainder operation on two arguments as prescribed by the IEEE 754 standard, which is not the same as the modulo operation computed by the % operator. The result of `remainder(x, y)` is $x - yn$, where n is the integer closest to the exact value of x/y . If two integers are equally close, then n is the integer that is even. This yields results that may be surprising, as indicated by the following examples:

```
>> remainder(1,2)
1.0
>> remainder(3,2)
-1.0
```

Compare this to

```
>> 3%2
1
```

which is different in two ways. The result numerically different and is of type *int*, whereas `remainder()` always yields a result of type *double*. The `remainder()` function is implemented by the `java.lang.Math` class, which calls it `IEEEremainder()`. The documentation for that class gives the following special cases:

- If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN.
- If the first argument is finite and the second argument is infinite, then the result is the same as the first argument.

DCT() and IDCT()

The DCT function can take one, two, or three arguments. In all three cases, the first argument is an array of length $N > 0$ and the DCT returns an

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos\left((2n+1)k\frac{\pi}{2D}\right) \quad (1)$$

for k from 0 to $D-1$, where N is the size of the specified array and D is the size of the DCT. If only one argument is given, then D is set to equal the next power of two larger than N . If a second argu-

ment is given, then its value is the *order* of the DCT, and the size of the DCT is 2^{order} . If a third argument is given, then it specifies the scaling factors s_k according to the following table:

TABLE 4: Normalization options for the DCT function

Name	Third argument	Normalization
Normalized	0	$s_k = \begin{cases} 1/\sqrt{2}; & k = 0 \\ 1; & \text{otherwise} \end{cases}$
Unnormalized	1	$s_k = 1$
Orthonormal	2	$s_k = \begin{cases} 1/\sqrt{D}; & k = 0 \\ \sqrt{2/D}; & \text{otherwise} \end{cases}$

The default, if a third argument is not given, is “Normalized.”

The IDCT function is similar, and can also take one, two, or three arguments. The formula in this case is

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos\left((2n+1)k\frac{\pi}{2D}\right). \quad (2)$$

5.8 Fixed Point Numbers

VisualSense includes a preliminary fixed point data type. We represent a fixed point value in the expression language using the following format:

```
fix(value, totalBits, integerBits)
```

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

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

The value can also be a matrix of doubles. The values are rounded, yielding the nearest value representable with the specified precision. If the value to represent is out of range, then it is saturated, meaning that the maximum or minimum fixed point value is returned, depending on the sign of the specified value. For example,

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

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

In addition to the `fix()` function, the expression language offers a `quantize()` function. The arguments are the same as those of the `fix()` function, but the return type is a `DoubleToken` or `DoubleMa-`

trixToken instead of a FixToken or FixMatrixToken. This function can therefore be used to quantize double-precision values without ever explicitly working with the fixed-point representation.

To make the FixToken accessible within the expression language, the following functions are available:

- To create a single FixPoint Token using the expression language:

```
fix(5.34, 10, 4)
```

This will create a FixToken. In this case, we try to fit the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with FixPoint values using the expression language:

```
fix([ -.040609, -.001628, .17853 ], 10, 2)
```

This will create a FixMatrixToken with 1 row and 3 columns, in which each element is a FixPoint value with precision(10/2). The resulting FixMatrixToken will try to fit each element of the given double matrix into a 10 bit representation with 2 bits used for the integer part. By default the round quantizer is used.

- To create a single DoubleToken, which is the quantized version of the double value given, using the expression language:

```
quantize(5.34, 10, 4)
```

This will create a DoubleToken. The resulting DoubleToken contains the double value obtained by fitting the number 5.34 into a 10 bit representation with 4 bits used in the integer part. This may lead to quantization errors. By default the round quantizer is used.

- To create a Matrix with doubles quantized to a particular precision using the expression language:

```
quantize([ -.040609, -.001628, .17853 ], 10, 2)
```

This will create a DoubleMatrixToken with 1 row and 3 columns. The elements of the token are obtained by fitting the given matrix elements into a 10 bit representation with 2 bits used for the integer part. Instead of being a fixed point value, the values are converted back to their double representation and by default the round quantizer is used.

Appendix A: Tables of Functions

In this appendix, we tabulate the functions available in the expression language. Further explanation of many of these functions is given in section section 5.7 above.

A.1 Trigonometric Functions

TABLE 5: Trigonometric functions.

function	argument type(s)	return type	description
acos	<i>double</i> in the range [-1.0, 1.0] or <i>complex</i>	<i>double</i> in the range [0.0, pi] or NaN if out of range or <i>complex</i>	arc cosine <i>complex</i> case: $\text{acos}(z) = -i \log(z + i \sqrt{1 - z^2})$
asin	<i>double</i> in the range [-1.0, 1.0] or <i>complex</i>	<i>double</i> in the range [-pi/2, pi/2] or NaN if out of range or <i>complex</i>	arc sine <i>complex</i> case: $\text{asin}(z) = -i \log(iz + \sqrt{1 - z^2})$
atan	<i>double</i> or <i>complex</i>	<i>double</i> in the range [-pi/2, pi/2] or <i>complex</i>	arc tangent <i>complex</i> case: $\text{atan}(z) = \frac{-i}{2} \log\left(\frac{i-z}{i+z}\right)$
atan2	<i>double, double</i>	<i>double</i> in the range [-pi, pi]	angle of a vector (note: the arguments are (y,x), not (x,y) as one might expect).
acosh	<i>double</i> greater than 1 or <i>complex</i>	<i>double</i> or <i>complex</i>	hyperbolic arc cosine, defined for both <i>double</i> and <i>complex</i> case by: $\text{acosh}(z) = \log(z + \sqrt{z^2 - 1})$
asinh	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	hyperbolic arc sine <i>complex</i> case: $\text{asinh}(z) = \log(z + \sqrt{z^2 + 1})$
cos	<i>double</i> or <i>complex</i>	<i>double</i> in the range [-1, 1], or <i>complex</i>	cosine <i>complex</i> case: $\text{cos}(z) = \frac{\exp(iz) + \exp(-iz)}{2}$
cosh	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	hyperbolic cosine, defined for <i>double</i> or <i>complex</i> by: $\text{cosh}(z) = \frac{\exp(z) + \exp(-z)}{2}$
sin	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	sine function <i>complex</i> case: $\text{sin}(z) = \frac{\exp(iz) - \exp(-iz)}{2i}$
sinh	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	hyperbolic sine, defined for <i>double</i> or <i>complex</i> by: $\text{sinh}(z) = \frac{\exp(z) - \exp(-z)}{2}$
tan	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	tangent function, defined for <i>double</i> or <i>complex</i> by: $\text{tan}(z) = \frac{\text{sin}(z)}{\text{cos}(z)}$
tanh	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	hyperbolic tangent, defined for <i>double</i> or <i>complex</i> by: $\text{tanh}(z) = \frac{\text{sinh}(z)}{\text{cosh}(z)}$

A.2 Basic Mathematical Functions

TABLE 6: Basic mathematical functions

function	argument type(s)	return type	description
abs	<i>double</i> or <i>int</i> or <i>long</i> or <i>complex</i>	<i>double</i> or <i>int</i> or <i>long</i> (<i>complex</i> returns <i>double</i>)	absolute value complex case: $\text{abs}(a + ib) = z = \sqrt{a^2 + b^2}$
angle	<i>complex</i>	<i>double</i> in the range $[-\pi, \pi]$	angle or argument of the complex number: $\angle z$
ceil	<i>double</i>	<i>double</i>	ceiling function, which returns the smallest (closest to negative infinity) <i>double</i> value that is not less than the argument and is an integer.
compare	<i>double, double</i>	<i>int</i>	compare two numbers, returning -1, 0, or 1 if the first argument is less than, equal to, or greater than the second.
conjugate	<i>complex</i>	<i>complex</i>	complex conjugate
exp	<i>double</i> or <i>complex</i>	<i>double</i> in the range $[0.0, \text{infinity}]$ or <i>complex</i>	exponential function (e^{argument}) complex case: $e^{a+ib} = e^a(\cos(b) + i\sin(b))$
floor	<i>double</i>	<i>double</i>	floor function, which is the largest (closest to positive infinity) value not greater than the argument that is an integer.
gaussian	<i>double, double</i> or <i>double, double, int</i> , or <i>double, double, int, int</i>	<i>double</i> or { <i>double</i> } or [<i>double</i>]	one or more Gaussian random variables with the specified mean and standard deviation (see page 55).
imag	<i>complex</i>	<i>double</i>	imaginary part
isInfinite	<i>double</i>	<i>boolean</i>	return true if the argument is infinite
isNaN	<i>double</i>	<i>boolean</i>	return true if the argument is “not a number”
log	<i>double</i> or <i>complex</i>	<i>double</i> or <i>complex</i>	natural logarithm complex case: $\log(z) = \log(\text{abs}(z) + i\angle(z))$
log10	<i>double</i>	<i>double</i>	log base 10
log2	<i>double</i>	<i>double</i>	log base 2
max	<i>double, double</i> or <i>int, int</i> or <i>long, long</i> or <i>unsignedByte, unsignedByte</i> or { <i>double</i> } or { <i>int</i> } or { <i>long</i> } or { <i>unsignedByte</i> }	<i>double</i> or <i>int</i> or <i>long</i> or <i>unsignedByte</i>	maximum
min	<i>double, double</i> or <i>int, int</i> or <i>long, long</i> or <i>unsignedByte, unsignedByte</i> or { <i>double</i> } or { <i>int</i> } or { <i>long</i> } or { <i>unsignedByte</i> }	<i>double</i> or <i>int</i> or <i>long</i> or <i>unsignedByte</i>	minimum

TABLE 6: Basic mathematical functions

function	argument type(s)	return type	description
neighborhood	<i>type, type, double</i>	<i>boolean</i>	return true if the first argument is in the neighborhood of the second, meaning that the distance is less than or equal to the third argument. The first two arguments can be any type for which such a distance is defined. For composite types, arrays, records, and matrices, then return true if the first two arguments have the same structure, and each corresponding element is in the neighborhood.
pow	<i>double, double or complex, complex</i>	<i>double or complex</i>	first argument to the power of the second
random	no arguments or <i>int</i> or <i>int, int</i>	<i>double</i> or <i>{double}</i> or <i>[double]</i>	one or more random numbers between 0.0 and 1.0 (see page 55)
real	<i>complex</i>	<i>double</i>	real part
remainder	<i>double, double</i>	<i>double</i>	remainder after division, according to the IEEE 754 floating-point standard (see page 56).
round	<i>double</i>	<i>long</i>	round to the nearest <i>long</i> , choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0L. If the argument is out of range, the result is either MaxLong or MinLong, depending on the sign.
roundToInt	<i>double</i>	<i>int</i>	round to the nearest <i>int</i> , choosing the next greater integer when exactly in between, and throwing an exception if out of range. If the argument is NaN, the result is 0. If the argument is out of range, the result is either MaxInt or MinInt, depending on the sign.
sgn	<i>double</i>	<i>int</i>	-1 if the argument is negative, 1 otherwise
sqrt	<i>double or complex</i>	<i>double or complex</i>	square root. If the argument is <i>double</i> with value less than zero, then the result is NaN. complex case: $\text{sqrt}(z) = \sqrt{ z } \left(\cos\left(\frac{\angle z}{2}\right) + i \sin\left(\frac{\angle z}{2}\right) \right)$
toDegrees	<i>double</i>	<i>double</i>	convert radians to degrees
toRadians	<i>double</i>	<i>double</i>	convert degrees to radians

A.3 Matrix, Array, and Record Functions.

TABLE 7: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
arrayToMatrix	{ <i>type</i> }, int, int	[<i>type</i>]	Create a matrix from the specified array with the specified number of rows and columns
conjugateTranspose	[<i>complex</i>]	[<i>complex</i>]	Return the conjugate transpose of the specified matrix.
createSequence	<i>type</i> , <i>type</i> , int	{ <i>type</i> }	Create an array with values starting with the first argument, incremented by the second argument, of length given by the third argument.
crop	[<i>int</i>], <i>int</i> , <i>int</i> , <i>int</i> , <i>int</i> or [<i>double</i>], <i>int</i> , <i>int</i> , <i>int</i> , <i>int</i> or [<i>complex</i>], <i>int</i> , <i>int</i> , <i>int</i> , <i>int</i> or [<i>long</i>], <i>int</i> , <i>int</i> , <i>int</i> , <i>int</i> or	[<i>int</i>] or [<i>double</i>] or [<i>complex</i>] or [<i>long</i>] or	Given a matrix of any <i>type</i> , return a submatrix starting at the specified row and column with the specified number of rows and columns.
determinant	[<i>double</i>] or [<i>complex</i>]	<i>double</i> or <i>complex</i>	Return the determinant of the specified matrix.
diag	{ <i>type</i> }	[<i>type</i>]	Return a diagonal matrix with the values along the diagonal given by the specified array.
divideElements	[<i>type</i>], [<i>type</i>]	[<i>type</i>]	Return the element-by-element division of two matrices
hilbert	<i>int</i>	[<i>double</i>]	Return a square Hilbert matrix, where $A_{ij} = 1/(i + j + 1)$. A Hilbert matrix is nearly, but not quite singular.
identityMatrixComplex	<i>int</i>	[<i>complex</i>]	Return an identity matrix with the specified dimension.
identityMatrixDouble	<i>int</i>	[<i>double</i>]	Return an identity matrix with the specified dimension.
identityMatrixInt	<i>int</i>	[<i>int</i>]	Return an identity matrix with the specified dimension.
identityMatrixLong	<i>int</i>	[<i>long</i>]	Return an identity matrix with the specified dimension.
intersect	<i>record</i> , <i>record</i>	<i>record</i>	Return a record that contains only fields that are present in both arguments, where the value of the field is taken from the first record.
inverse	[<i>double</i>] or [<i>complex</i>]	[<i>double</i>] or [<i>complex</i>]	Return the inverse of the specified matrix, or throw an exception if it is singular.
matrixToArray	[<i>type</i>]	{ <i>type</i> }	Create an array containing the values in the matrix
merge	<i>record</i> , <i>record</i>	<i>record</i>	Merge two records, giving priority to the first one when they have matching record labels.
multiplyElements	[<i>type</i>], [<i>type</i>]	[<i>type</i>]	Multiply elementwise the two specified matrices.
orthogonalizeColumns	[<i>double</i>] or [<i>complex</i>]	[<i>double</i>] or [<i>complex</i>]	Return a similar matrix with orthogonal columns.
orthogonalizeRows	[<i>double</i>] or [<i>complex</i>]	[<i>double</i>] or [<i>complex</i>]	Return a similar matrix with orthogonal rows.
orthonormalizeColumns	[<i>double</i>] or [<i>complex</i>]	[<i>double</i>] or [<i>complex</i>]	Return a similar matrix with orthonormal columns.
orthonormalizeRows	[<i>double</i>] or [<i>complex</i>]	[<i>double</i>] or [<i>complex</i>]	Return a similar matrix with orthonormal rows.
repeat	<i>int</i> , <i>type</i>	{ <i>type</i> }	Create an array by repeating the specified token the specified number of times.
sum	{ <i>type</i> } or [<i>type</i>]	<i>type</i>	Sum the elements of the specified array or matrix. This throws an exception if the elements do not support addition or if the array is empty (an empty matrix will return zero).
trace	[<i>type</i>]	<i>type</i>	Return the trace of the specified matrix.

TABLE 7: Functions that take or return matrices, arrays, or records.

function	argument type(s)	return type	description
transpose	[<i>type</i>]	[<i>type</i>]	Return the transpose of the specified matrix.
zeroMatrixComplex	<i>int, int</i>	[<i>complex</i>]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixDouble	<i>int, int</i>	[<i>double</i>]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixInt	<i>int, int</i>	[<i>int</i>]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixLong	<i>int, int</i>	[<i>long</i>]	Return a zero matrix with the specified number of rows and columns.

A.4 Functions for Evaluating Expressions

TABLE 8: Utility functions for evaluating expressions

function	argument type(s)	return type	description
eval	<i>string</i>	any type	evaluate the specified expression (see page 55).
parseInt	<i>string</i> or <i>string, int</i>	<i>int</i>	return an <i>int</i> read from a <i>string</i> , using the given radix if a second argument is provided.
parseLong	<i>string</i> or <i>string, int</i>	<i>int</i>	return a <i>long</i> read from a <i>string</i> , using the given radix if a second argument is provided.
toBinaryString	<i>int</i> or <i>long</i>	<i>string</i>	return a binary representation of the argument
toOctalString	<i>int</i> or <i>long</i>	<i>string</i>	return an octal representation of the argument
toString	<i>double</i> or <i>int</i> or <i>int, int</i> or <i>long</i> or <i>long, int</i>	<i>string</i>	return a string representation of the argument, using the given radix if a second argument is provided.
traceEvaluation	<i>string</i>	<i>string</i>	evaluate the specified expression and report details on how it was evaluated (see page 55).

A.5 Signal Processing Functions

TABLE 9: Functions performing signal processing operations

function	argument type(s)	return type	description
convolve	{double}, {double} or {complex}, {complex}	{double} or {complex}	Convolve two arrays and return an array whose length is sum of the lengths of the two arguments minus one. Convolution of two arrays is the same as polynomial multiplication.
DCT	{double} or {double}, int or {double}, int, int	{double}	Return the discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 56).
downsample	{double}, int or {double}, int, int	{double}	Return a new array with every n -th element of the argument array, where n is the second argument. If a third argument is given, then it must be between 0 and $n - 1$, and it specifies an offset into the array (by giving the index of the first output).
FFT	{double} or {complex} or {double}, int {complex}, int	{complex}	Return the fast Fourier transform of the specified array. If the second argument is given with value n , then the length of the transform is 2^n . Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
generateBartlettWindow	int	{double}	Return a Bartlett (rectangular) window with the specified length. The end points have value 0.0, and if the length is odd, the center point has value 1.0. For length $M + 1$, the formula is: $w(n) = \begin{cases} 2\frac{n}{M}, & \text{if } 0 \leq n \leq \frac{M}{2} \\ 2 - 2\frac{n}{M}, & \text{if } \frac{M}{2} \leq n \leq M \end{cases}$
generateBlackmanWindow	int	{double}	Return a Blackman window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.42 + 0.5 \cos(2\pi n/M) + 0.08 \cos(4\pi n/M)$
generateBlackmanHarrisWindow	int	{double}	Return a Blackman-Harris window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.35875 + 0.48829 \cos(2\pi n/M) + 0.14128 \cos(4\pi n/M) + 0.01168 \cos(6\pi n/M)$
generateGaussianCurve	double, double, int	{double}	Return a Gaussian curve with the specified standard deviation, extent, and length. The extent is a multiple of the standard deviation. For instance, to get 100 samples of a Gaussian curve with standard deviation 1.0 out to four standard deviations, use generateGaussianCurve(1.0, 4.0, 100).
generateHammingWindow	int	{double}	Return a Hamming window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.54 - 0.46 \cos(2\pi n/M)$
generateHanningWindow	int	{double}	Return a Hanning window with the specified length. For length $M + 1$, the formula is: $w(n) = 0.5 - 0.5 \cos(2\pi n/M)$

TABLE 9: Functions performing signal processing operations

function	argument type(s)	return type	description
generatePolynomialCurve	{double}, double, double, int	{double}	Return samples of a curve specified by a polynomial. The first argument is an array with the polynomial coefficients, beginning with the constant term, the linear term, the squared term, etc. The second argument is the value of the polynomial variable at which to begin, and the third argument is the increment on this variable for each successive sample. The final argument is the length of the returned array.
generateRaisedCosinePulse	double, double, int	{double}	Return an array containing a symmetric raised-cosine pulse. This pulse is widely used in communication systems, and is called a “raised cosine pulse” because the magnitude its Fourier transform has a shape that ranges from rectangular (if the excess bandwidth is zero) to a cosine curved that has been raised to be non-negative (for excess bandwidth of 1.0). The elements of the returned array are samples of the function: $h(t) = \frac{\sin(\pi t/T)}{\pi t/T} \times \frac{\cos(x\pi t/T)}{1 - (2xt/T)^2},$ where x is the excess bandwidth (the first argument) and T is the number of samples from the center of the pulse to the first zero crossing (the second argument). The samples are taken with a sampling interval of 1.0, and the returned array is symmetric and has a length equal to the third argument. With an excessBandwidth of 0.0, this pulse is a sinc pulse.
generateRectangularWindow	int	{double}	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	{double} or {double}, int or {double}, int, int	{double}	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see page 56).
IFFT	{double} or {complex} or {double}, int {complex}, int	{complex}	Return the inverse fast Fourier transform of the specified array. If the second argument is given with value n , then the length of the transform is 2^n . Otherwise, the length is the next power of two greater than or equal to the length of the input array. If the input length does not match this length, then input is padded with zeros.
nextPowerOfTwo	double	int	Return the next power of two larger than or equal to the argument.
poleZeroToFrequency	{complex}, {complex}, complex, int	{complex}	Given an array of pole locations, an array of zero locations, a gain term, and a size, return an array of the specified size representing the frequency response specified by these poles, zeros, and gain. This is calculated by walking around the unit circle and forming the product of the distances to the zeros, dividing by the product of the distances to the poles, and multiplying by the gain.
sinc	double	double	Return the sinc function, $\sin(x)/x$, where special care is taken to ensure that 1.0 is returned if the argument is 0.0.

TABLE 9: Functions performing signal processing operations

function	argument type(s)	return type	description
toDecibels	<i>double</i>	<i>double</i>	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	{ <i>double</i> }	{ <i>double</i> }	Modify the specified array to unwrap the angles. That is, if the difference between successive values is greater than π in magnitude, then the second value is modified by multiples of 2π until the difference is less than or equal to π . In addition, the first element is modified so that its difference from zero is less than or equal to π in magnitude.
upsample	{ <i>double</i> }, <i>int</i>	{ <i>double</i> }	Return a new array that is the result of inserting $n - 1$ zeroes between each successive sample in the input array, where n is the second argument. The returned array has length nL , where L is the length of the argument array. It is required that $n > 0$.

A.6 I/O Functions and Other Miscellaneous Functions

TABLE 10: Miscellaneous functions.

function	argument type(s)	return type	description
cast	<i>type1</i> , <i>type2</i>	<i>type1</i>	Return the second argument converted to the type of the first, or throw an exception if the conversion is invalid.
constants	none	<i>record</i>	Return a record identifying all the globally defined constants in the expression language.
findFile	<i>string</i>	<i>string</i>	Given a file name relative to the user directory, current directory, or classpath, return the absolute file name of the first match, or return the name unchanged if no match is found.
freeMemory	none	<i>long</i>	Return the approximate number of bytes available for future memory allocation.
iterate	<i>function</i> , <i>int</i> , <i>type</i>	{ <i>type</i> }	Return an array that results from first applying the specified function to the third argument, then applying it to the result of that application, and repeating to get an array whose length is given by the second argument.
map	<i>function</i> , { <i>type</i> }	{ <i>type</i> }	Return an array that results from applying the specified function to the elements of the specified array.
property	<i>string</i>	<i>string</i>	Return a system property with the specified name from the environment, or an empty string if there is none. Some useful properties are java.version, ptolomy.ptII.dir, ptolomy.ptII.dirAsURL, and user.dir.
readFile	<i>string</i>	<i>string</i>	Get the string text in the specified file, or throw an exception if the file cannot be found. The file can be absolute, or relative to the current working directory (user.dir), the user's home directory (user.home), or the classpath.
readResource	<i>string</i>	<i>string</i>	Get the string text in the specified resource (which is a file found relative to the classpath), or throw an exception if the file cannot be found.
totalMemory	none	<i>long</i>	Return the approximate number of bytes used by current objects plus those available for future object allocation.

Appendix B: References

- [1] P. Baldwin, "Sensor Networks Modeling and Simulation in Ptolemy II," Unnumbered Technical Report, UC Berkeley, August 8, 2003.
- [2] A. Cataldo, C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer and H. Zheng, "Hyvisual: A Hybrid System Visual Modeler," Technical Memorandum UCB/ERL M03/30, University of California, Berkeley, July 17, 2003.
- [3] C. T. Ee, N. V. Krishnan and S. Kohli, "Efficient Broadcasts in Sensor Networks," Unpublished Class Project Report, UC Berkeley, Berkeley, CA, May 12, 2003.
- [4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong, "Taming Heterogeneity-the Ptolemy Approach," *Proceedings of the IEEE*, **91**(2), January, 2003.
- [5] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos and D. Estrin, "Emstar: An Environment for Developing Wireless Embedded Systems Software," CENS Technical Report 0009, Center for Embedded Networked Sensing, UCLA, March 24, 2003.
- [6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
- [7] A. Girault, B. Lee and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18**(6), June 1999.
- [8] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java: Volume 1: Introduction to Ptolemy II," Technical Memorandum UCB/ERL M03/27, University of California, Berkeley, CA USA 94720, July 16, 2003.
- [9] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java: Volume 3: Ptolemy II Domains," Technical Memorandum UCB/ERL M03/29, University of California, Berkeley, CA USA 94720, July 16, 2003.
- [10] T. J. Kwon and M. Geria, "Efficient Flooding with Passive Clustering (Pc) in Ad Hoc Networks," *ACM SIGCOMM Computer Communication Review*, **32**(1), January, 2002.
- [11] E. A. Lee, "Modeling Concurrent Real-Time Processes Using Discrete Events," *Annals of Software Engineering* **7**: 25-45, March 4th 1998.
- [12] M. Löbbers, D. Willkomm, A. Köpke, H. Karl, "Framework for Simulation of Mobility in OMNeT++ (Mobility Framework)," February 09 2004, http://www.tkn.tu-berlin.de/research/research_texte/framework.html.
- [13] The CMU Monarch Project, "The CMU Monarch Project's Wireless and Mobility Extensions to NS," 1998 (see also <http://www.monarch.cs.cmu.edu/>)
- [14] Ns-2, <http://www.isi.edu/nsnam/ns>, 2004.

- [15] OPNET Technologies, Inc., “OPNET Modeler,” <http://opnet.com/products/modeler/home.html>, 2004.
- [16] S. Park, A. Savvides and M. B. Srivastava, “Sensorsim: A Simulation Framework for Sensor Networks,” *3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, Massachusetts, United States, ACM Press, August 20, 2000 (see also <http://nesl.ee.ucla.edu/projects/sensorsim/>).
- [17] H.-Y. Tyan, “Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation,” Ph.D. Dissertation, Ohio State University, 2002. (see also <http://www.j-sim.org>)
- [18] A. Varga, “The Omnet++ Discrete Event Simulation System,” *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 6-9, 2001 [see also <http://www.omnetpp.org/>].
- [19] Y. Xiong, “An Extensible Type System for Component-Based Design,” Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002.