This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

First Edition, Version 1.0

Please cite this book as:

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

Part III

Modeling Infrastructure

This part of this book focuses on the modeling infrastructure provided by Ptolemy II. Chapter 12 provides an overview of the software architecture with the goal of providing the reader with a good starting point for creating extensions of Ptolemy II. Chapter 13 describes the expression language used to set parameter values and perform computation in the Expression actor. Chapter 17 gives an overview of signal plotting capabilities provided in the actor library. Chapter 14 describes the Ptolemy II type system. Chapter 15 describes the ontology system. and Chapter 16 describes interfaces to the web, including mechanisms for exporting Ptolemy models to websites and mechanisms for creating custom web servers.

12 Software Architecture

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

Contents

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

12.1 Package Structure

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

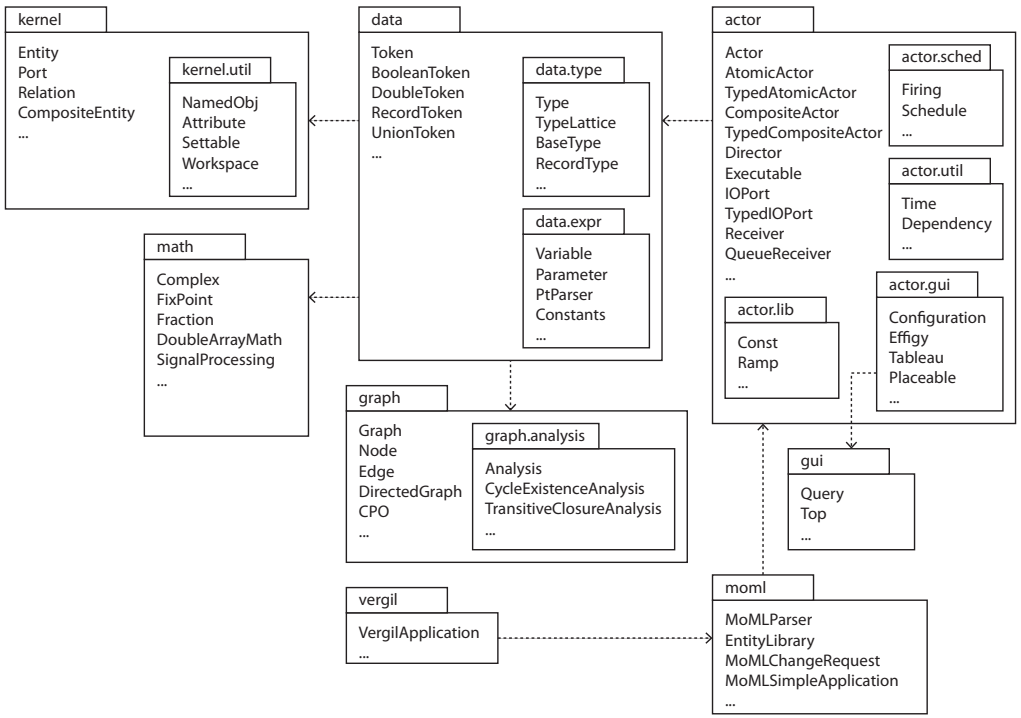


Figure 12.1: Key packages and their classes in Ptolemy II.

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

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

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

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

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

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

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

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

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

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

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

12.2 The Structure of Models

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

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

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

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

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

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

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

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

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

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

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

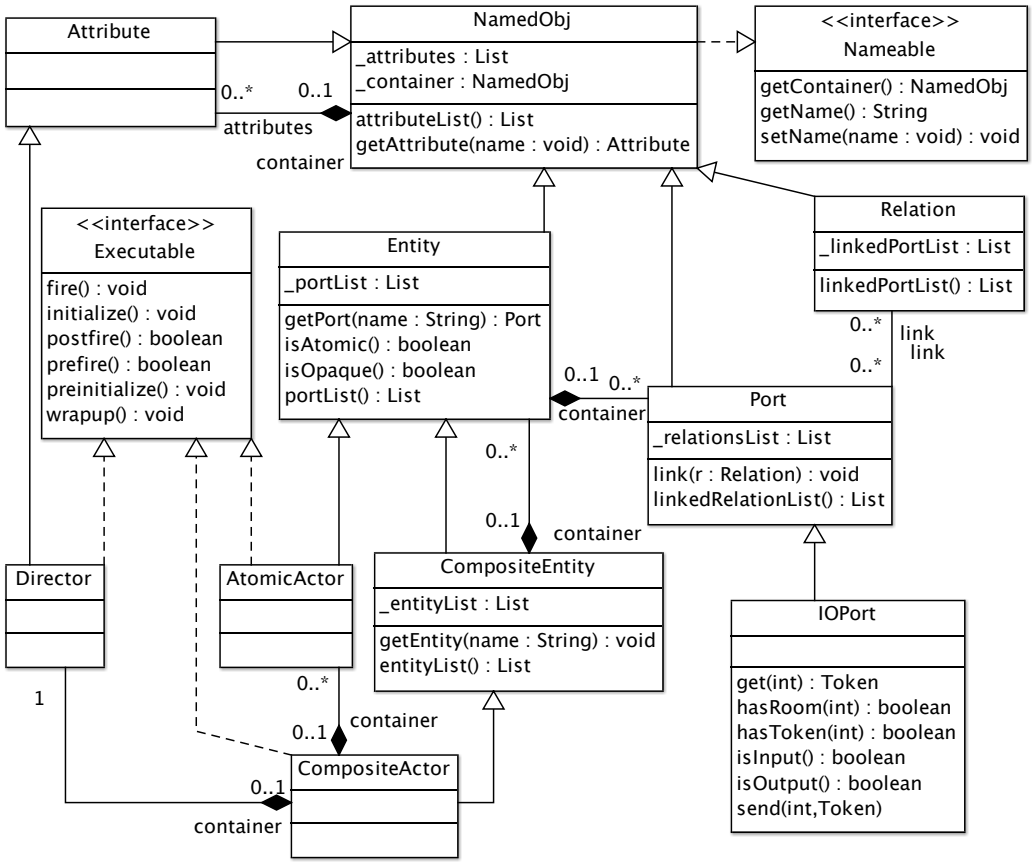


Figure 12.2: A meta model for Ptolemy II. This is a UML class diagram. The boxes represent classes with the class name, key members, and key methods shown. The lines with triangular arrowheads represent inheritance. The lines with diamond ends represent containment.

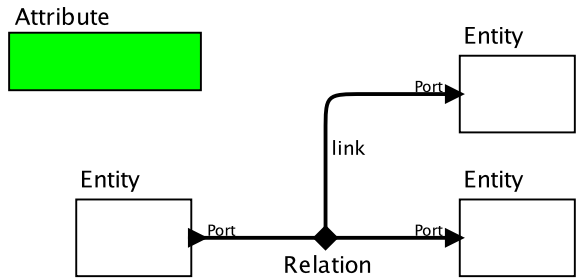


Figure 12.3: A Ptolemy II model showing the base meta-model class names for the objects in the model.

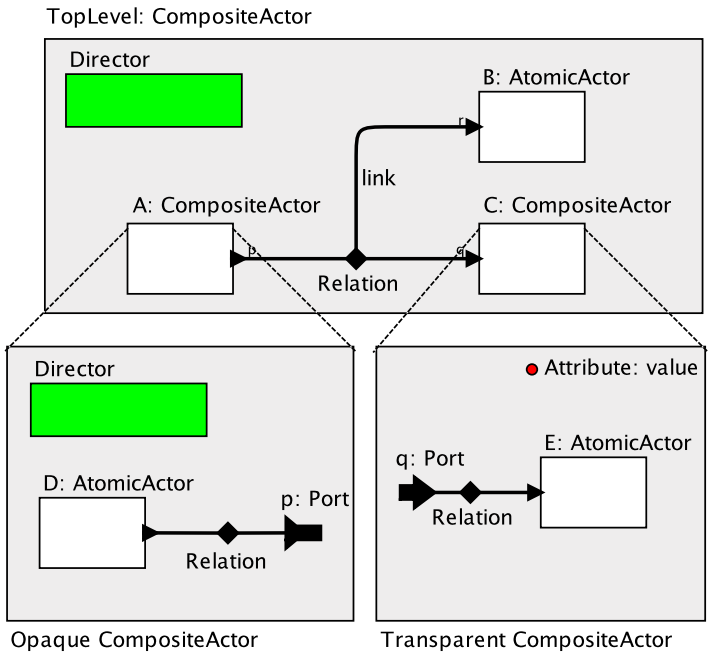


Figure 12.4: A hierarchical model showing meta-model class names for the objects in the model.

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

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

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

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

12.3 Actor Semantics and the MoC

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

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

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

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

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="TopLevel" class="ptolemy.actor.CompositeActor">
    <property name="Director" class="ptolemy.actor.Director"/>
    <entity name="A" class="ptolemy.actor.CompositeActor">
        <property name="Director" class="ptolemy.actor.Director"/>
        <port name="p" class="ptolemy.actor.IOPort"/>
        <entity name="D" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="p" relation="r"/>
        <link port="D.p" relation="r"/>
    </entity>
    <entity name="B" class="ptolemy.actor.AtomicActor">
        <port name="r" class="ptolemy.actor.IOPort"/>
    </entity>
    <entity name="C" class="ptolemy.actor.CompositeActor">
        <property name="Attribute"
                  class="ptolemy.kernel.util.Attribute"/>
        <port name="q" class="ptolemy.actor.IOPort"/>
        <entity name="E" class="ptolemy.actor.AtomicActor">
            <port name="p" class="ptolemy.actor.IOPort"/>
        </entity>
        <relation name="r" class="ptolemy.actor.IORelation"/>
        <link port="q" relation="r"/>
        <link port="E.p" relation="r"/>
    </entity>
    <relation name="r" class="ptolemy.actor.IORelation"/>
    <link port="A.p" relation="r"/>
    <link port="B.r" relation="r"/>
    <link port="C.q" relation="r"/>
</entity>
```

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

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

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

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

12.3.1 Execution Control

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

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

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

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

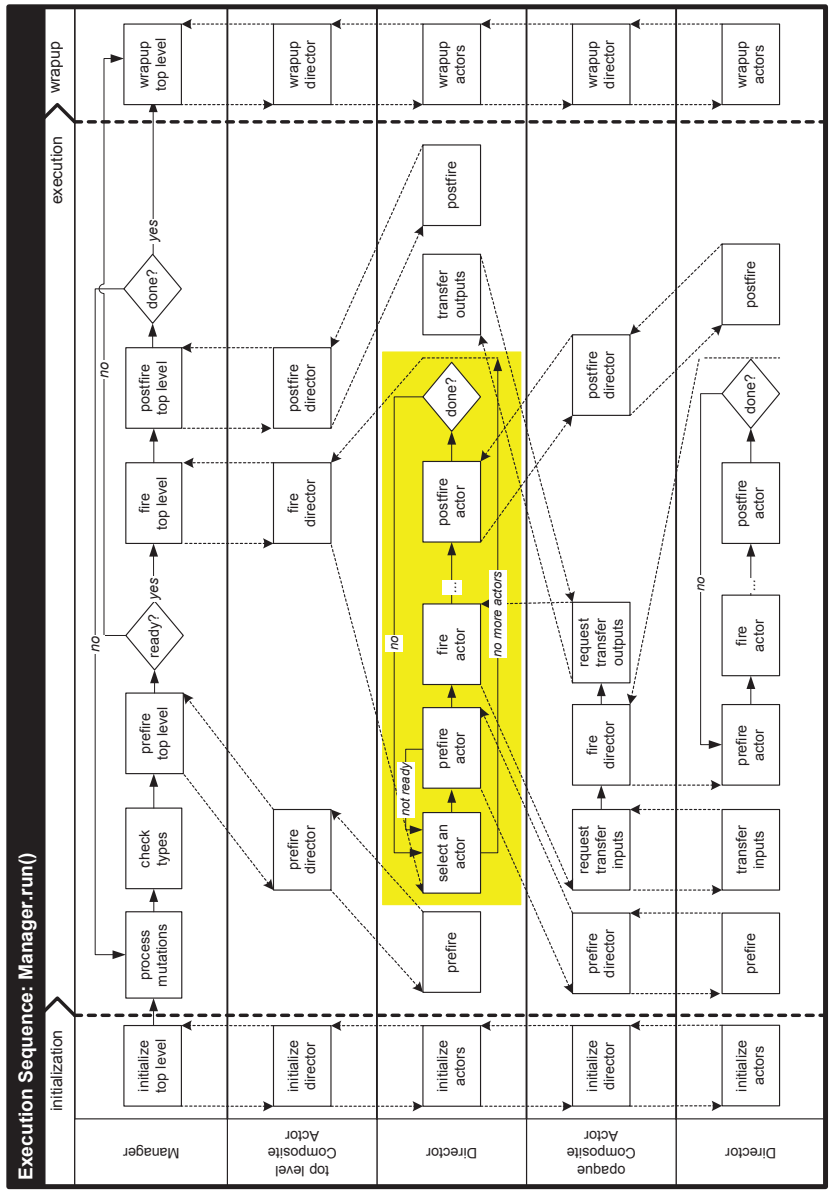


Figure 12.7: Execution of a hierarchical model with an opaque composite actor.

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

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

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

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

12.3.2 Communication

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

Sidebar: Why prefire, fire, and postfire?

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

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

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

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

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

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

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

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

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

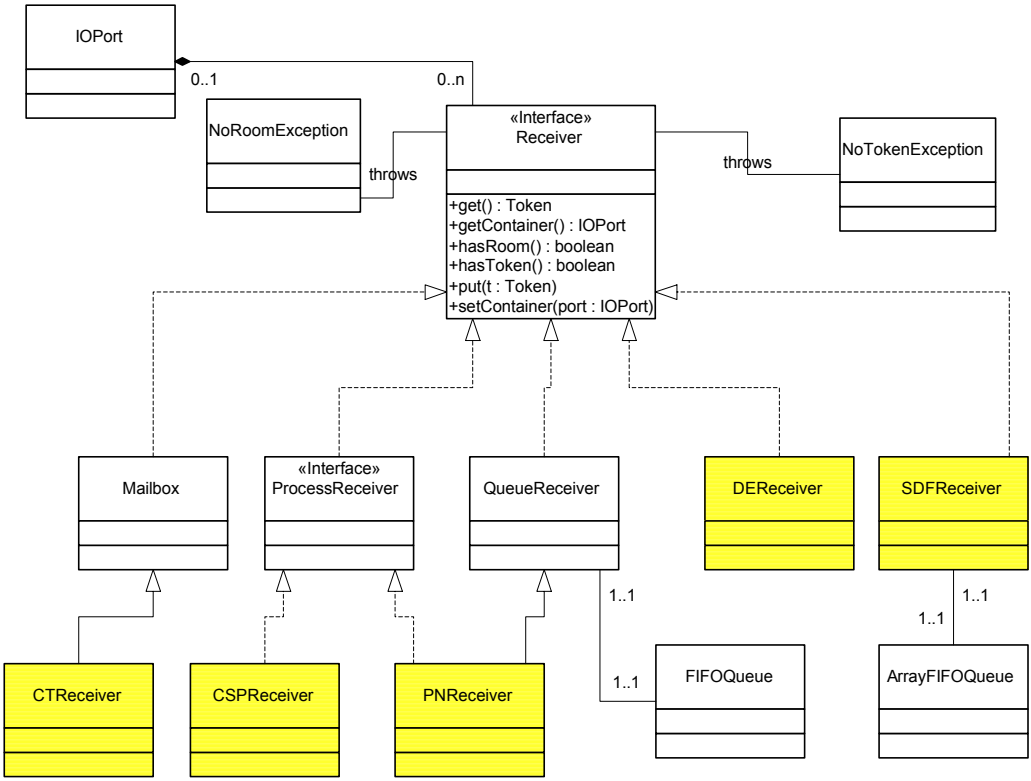


Figure 12.8: Meta model for communication in Ptolemy II.

12.3.3 Time

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

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

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

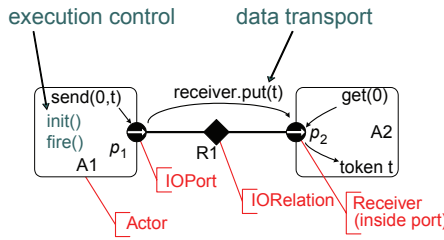


Figure 12.9: Communication mechanism in Ptolemy II.

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

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

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

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

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

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

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

12.4 Designing Actors in Java[†]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 12.10: A simple Count actor.

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

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

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

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

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

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

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

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

12.4.1 Ports

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

Multiports and Single Ports

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

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

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

Reading and Writing

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

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

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

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

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

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

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

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

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

A token can be read from a channel by

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

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

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

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

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

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

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

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

Dependencies Between Ports

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

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

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

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

Port Production and Consumption Rates

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

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

12.4.2 Parameters

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

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

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

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

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

12.4.3 Coupled Port and Parameter

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

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

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

12.5 Summary

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

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

Figure 12.13: Code segments from the Ramp actor.

13 Expressions

Christopher Brooks, Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Neil Smyth, Yuhong Xiong

Contents

In Ptolemy II, 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 Ptolemy II **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, and it can be used by programmers extending the Ptolemy II system. In this chapter, we describe how to use expressions from the perspective of a user rather than a programmer.

Vergil provides an interactive **expression evaluator**, which is accessed through the menu command [`File→New→Expression Evaluator`]. This operates like an interactive command shell, and is shown in Figure 13.1. 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.

13.1 Simple Arithmetic Expressions

13.1.1 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`, `MaxShort`, `MinShort`, `MaxInt`, `MinInt`, `MaxLong`, `MinLong`, `MaxFloat`, `MinFloat`, `MaxDouble`, and `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 $\sqrt{-1}$. 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 followed by "`f`" or "`F`" are of type *float*. Numerical values without decimal points followed by the character "`l`" (el) or "`L`" are of type *long*. long. Numerical values without decimal points followed by the character "`s`" or "`S`" are of type *short*. 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 is not quite the same as the Java byte, which has a value between -128 and 127. Numbers of type *int*, *long*, *short* 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.

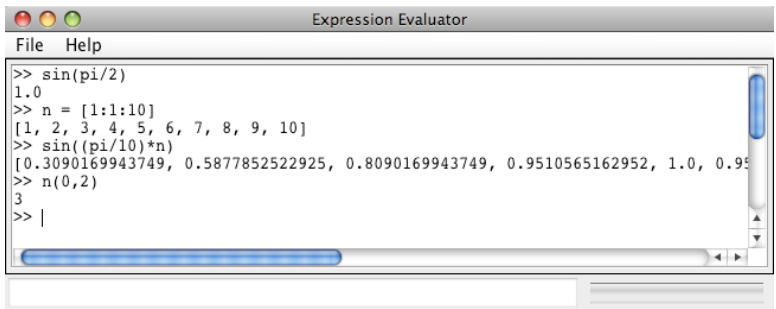


Figure 13.1: The Expression Evaluator.

A *complex* is defined by appending an "`i`" or a "`j`" to a *double* for the imaginary part. This gives a purely imaginary *complex* number which can then leverage the polymorphic operations in the Token classes to create a general *complex* number. Thus `2 + 3i` will result in the expected *complex* number. You can optionally write this `2 + 3*i`.

Literal string constants are also supported. Anything between *double* quotation marks, `"..."`, is interpreted as a *string* constant. The following built-in string-valued constants are defined:

variable name	meaning	JVM property name	example value
`PTII`	The directory in which Ptolemy II 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`
`TMPDIR`	The temporary directory	`java.io.tmpdir`	`c:\Documents and Settings\you\Local Settings\Temp\`
`USERNAME`	The user account name	`user.name`	`ptolemy`

The value of these variables is the value given by the corresponding Java virtual machine (**JVM**) property, such as `user.home` for `HOME`. The properties `user.dir` and `user.home` are standard in Java. Their values are platform dependent; see the documentation for the method `getProperties` in the java.lang.System class for details.* Vergil will display all the Java properties if you invoke [`View→JVM Properties`] in the menu of a Graph Editor.

The `ptolemy.ptII.dir` property is set automatically when Vergil or any other Ptolemy II executable is started up. You can also set it when you start a Ptolemy II process using the `java` command by a syntax like the following:

```
java -Dptolemy.ptII.dir=${PTII} classname
```

where `classname` is the full class name of a Java application. You can similarly set the other variables in the table. For example, to invoke Vergil in a particular directory, use

*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.

variable name	value	variable name	value
CLASSPATH	”xxxxxxCLASSPATHxxxxxx”	CWD	”/Users/eal”
E	2.718281828459	HOME	”/Users/eal”
Infinity	Infinity	MaxDouble	1.797693134862316E308
MaxFloat	3.402823466385289E38	MaxInt	2147483647
MaxLong	9223372036854775807L	MaxShort	32767s
MaxUnsignedByte	255ub	MinDouble	4.9E-324
MinFloat	1.401298464324817E-45	MinInt	-2147483648
MinLong	-92233720368547758O8L	MinShort	-32768s
MinUnsignedByte	0ub	NaN	NaN
NegativeInfinity	-Infinity	PI	3.1415926535898
PTII	”/ptII”	PositiveInfinity	Infinity
TMPDIR	”/tmp”	USERNAME	”eal”
backgroundColor	{0.9, 0.9, 0.9, 1.0}	boolean	false
complex	0.0 + 0.0i	double	0.0
e	2.718281828459	false	false
fixedpoint	fix(0, 2, 2)	float	0.0f
general	present	i	0.0 + 1.0i
int	0	j	0.0 + 1.0i
long	0L	matrix	[]
nil	nil	null	object(null)
object	object(null)	pi	3.1415926535898
scalar	present	short	0s
string	””	true	true
unknown	present	unsignedByte	0ub
xmltoken	null		

Table 13.1: An example of the values returned by the `constants` function

```
java -cp /ptII -Duser.dir=/Users/eal \
      ptolemy.vergil.VergilApplication
```

The `-cp` option specifies the classpath (which has to include the root directory of Ptolemy II), the `-D` option specifies the property to set, and the final argument is the class that includes a `main` method that invokes Vergil.

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 similar to what is shown in Figure 13.1.

13.1.2 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" 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 Ptolemy II 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 [`Customize→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.

Occasionally, it is desirable to access parameters that are not in scope. The expression language supports a limited syntax that permits access to certain variables out of scope. In particular, if in place of a variable name `x` in an expression you write `A::x`, then instead of looking for `x` in scope, the interpreter looks for a container named `A` in the scope and a parameter named `x` in `A`. This allows reaching down one level in the hierarchy from either the current container or any of its containers.

13.1.3 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 a type that losslessly converts to an integer such as an *int*, *short*, or an *unsignedByte*.

The *unsignedByte*, *short*, *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`, since `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 13.6). 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 Section 13.4.8). For example, counter-intuitively,

```
>> 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, *short* can be converted to *int*, *int* can be converted to *double*, *float* 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.
```

Just as *long* cannot be cast to *double*, *int* cannot be cast to *float* and vice versa.

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 *int* and *fixedpoint* types, overflow results in wraparound. For instance, the value of `MaxInt` is `2147483647`, but the expression `MaxInt + 1` yields `-2147483648`. Similarly, while `MaxUnsignedByte` has value `255ub`, `MaxUnsignedByte + 1ub` has value `0ub`. 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*, *short*, *int* and *long* (but not *fixedpoint*, *float*, *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 if `x` is undefined.

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*, *short*, *int*, and *long*.

13.1.4 Comments

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

13.2 Uses of Expressions

Expressions are used in Ptolemy II to assign values to parameters, to specify the input-output function realized by an Expression actor, and to specify guards and actions in state machines.

13.2.1 Parameters

The values of most parameters of actors can be given as expressions[†]. 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, as explained below in section 13.7. Adding parameters to actors is straightforward, as explained in chapter 2.

13.2.2 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 13.2.

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→Rename`], as shown in Figure 13.2. In the figure, the name is set to "noiseLevel." Then set the default value by double clicking. 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→SequenceSources` library in Vergil. It is shown in Figure 13.3. 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.

[†] 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.

13.2.3 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 parameters in scope by name using the syntax $*name* or ${*name*} where *name* is the name of the parameter in scope. For example, the StringCompare actor in Figure 13.4 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.

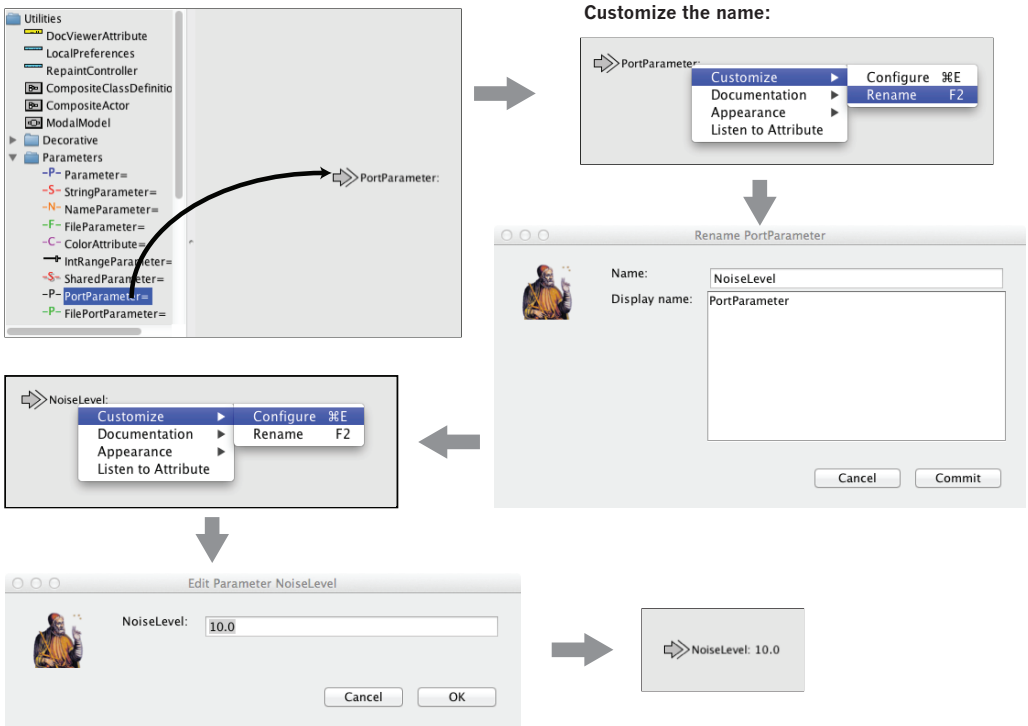


Figure 13.2: A *PortParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful and set its default value.

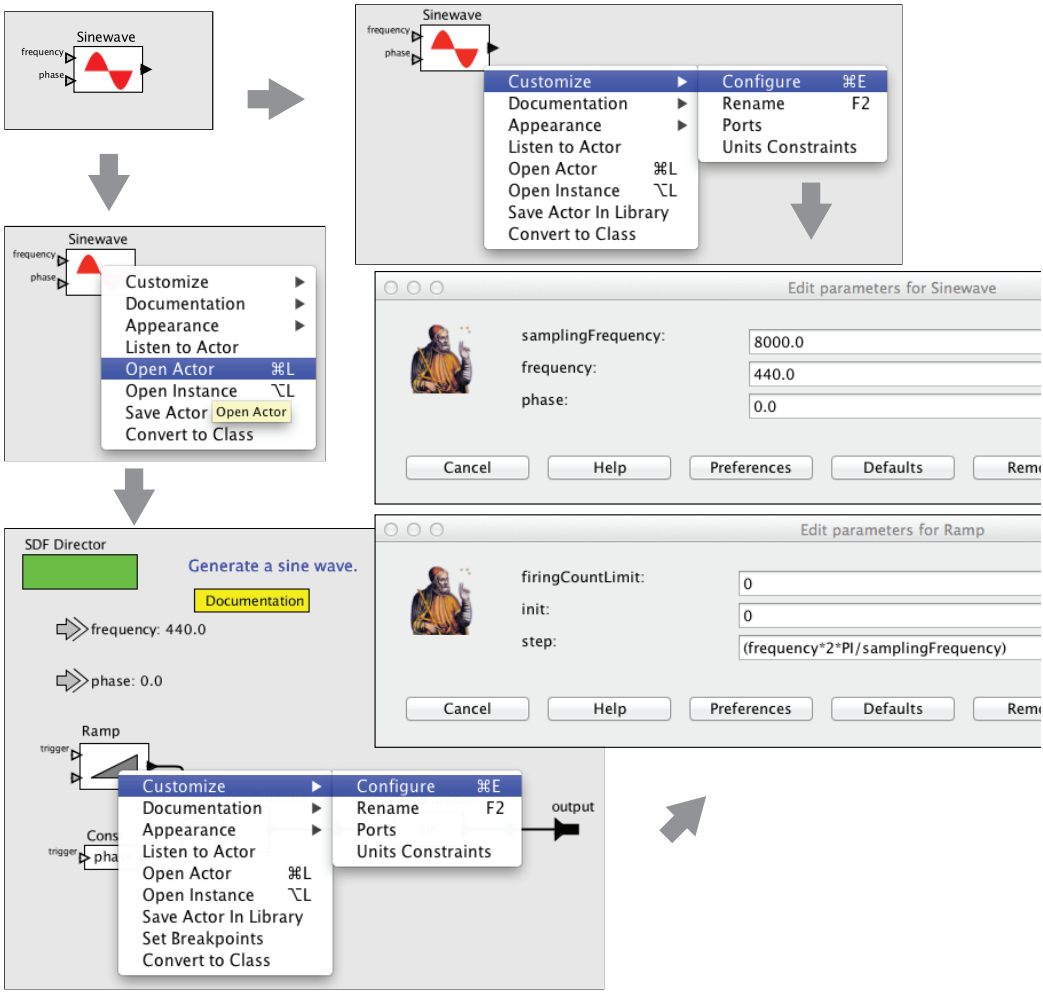


Figure 13.3: Sinewave actor, showing its port parameters, and their use at the lower level of hierarchy.

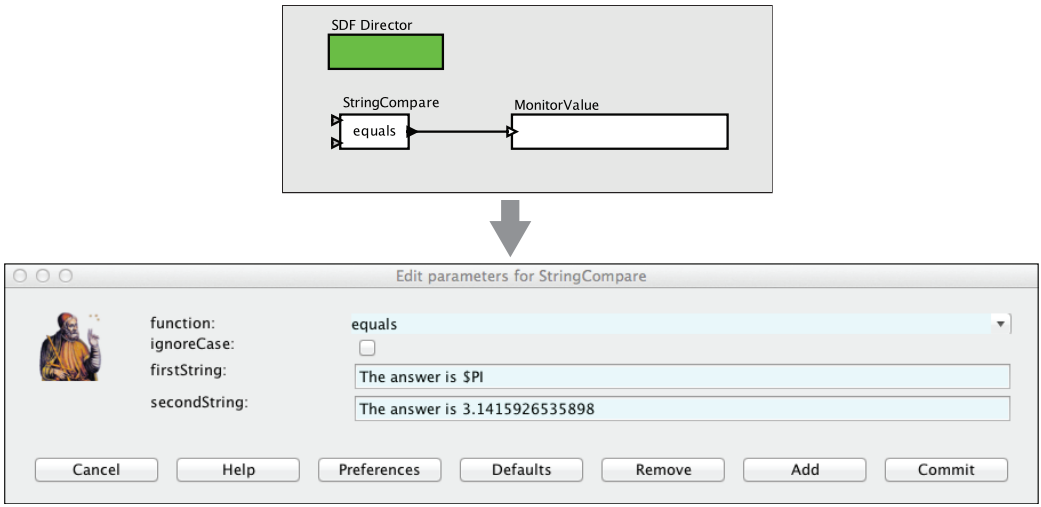


Figure 13.4: String parameters are indicated in the parameter editor boxes by a light blue background. A string parameter can include references to variables in scope with $*name*, where *name* is the name of the variable. In this example, the built-in constant $PI is referenced by name in the first parameter.

13.2.4 Expression Actor

The **Expression** actor is a particularly useful actor found in the `Math`. By default, it has one output and no inputs, as shown in Figure 13.5(a). The first step in using it is to add ports, as shown in (b) and (c). Click on Add to add a port, and then type in a unique name for the port. You then specify an expression using the port names as variables, as shown in (d), resulting in the icon shown in (e).

13.2.5 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.

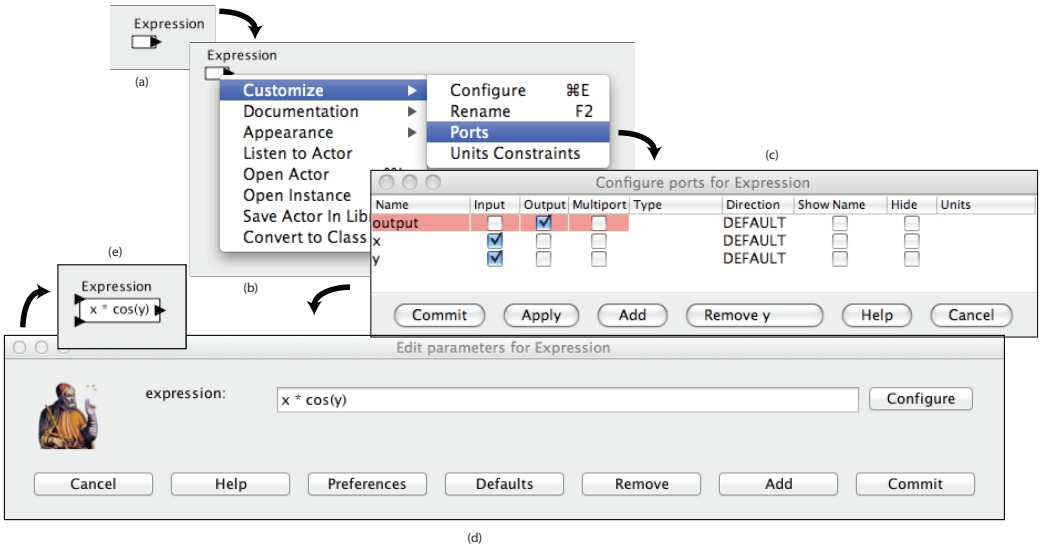


Figure 13.5: Illustration of the Expression actor.

13.3 Composite Data Types

A **composite data type** is a data type that aggregates some collection of other data types. The composite data types in the expression language include arrays, matrices, records, and union types.

13.3.1 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 type *string*. The types are denoted `arrayType(int,3)` and `arrayType(string,3)` respectively. An array is an ordered list of tokens of any type, with the primary constraint being that the elements all have 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

```
{1.0, 2.3}
```

Its type is `arrayType(double,2)`. The common type might be `arrayType(scalar)`, which is a union type (a type that can contain multiple distinct types). For example,

```
{1, 2.3, true}
```

has value

```
{1, 2.3, true}
```

The value is unchanged, but the type of the array is now `arrayType(scalar,3)`.

In Figure 13.5(c), the "Type" column may be used to specify the type of a port. Usually, it is not necessary to set the type, since the type inference mechanism will determine the type from the connections (see Chapter 14). Occasionally, however, it is necessary or helpful to force a port to have a particular type.

The Type column accepts expressions like `arrayType(int)`, which specifies an array with an unknown length. It is better, however, to specify an array length, if it is known. To do that, use an expression like `arrayType(int,`*n*`)`, where *n* is a positive integer that is the length of the array that is expected on the port.

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
```

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}
```

Addition, subtraction, multiplication, division, and modulo of arrays by scalars is also supported, as in the following examples:

```
>> {1.0, 2.0} / 2.0
{0.5, 1.0}
>> 1.0 / {2.0, 4.0}
{0.5, 0.25}
>> 3 * {2, 3}
{6, 9}
>> 12 / {3, 4}
{4, 3}
```

Arrays of length 1 are equivalent to scalars, as illustrated below:

```
>> {1.0, 2.0} / {2.0}
{0.5, 1.0}
>> {1.0} / {2.0, 4.0}
{0.5, 0.25}
>> {3} * {2, 3}
```

```
{6, 9}
>> {12} / {3, 4}
{4, 3}
```

A significant subtlety arises when using nested arrays. Note the following example:

```
>> {{1.0, 2.0}, {3.0, 1.0}} / {0.5, 2.0}
{{2.0, 4.0}, {1.5, 0.5}}
```

In this example, the left argument of the divide is an array with two elements, and the right argument is also an array with two elements. The divide is thus element wise. However, each division is the division of an array by a scalar.

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 13.5). 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
```

You can obtain the length of an array as follows,

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

You can extract a subarray by invoking the `subarray` method as follows:

```
>> {1, 2, 3, 4}.subarray(2, 2)
{3, 4}
```

The first argument is the starting index of the subarray, and the second argument is the length.

You can also extract non-contiguous elements from an array using the `extract` method. This method has two forms. The first form takes a *boolean* array of the same length as the original array which indicates which elements to extract, as in the following example:

```
>> {"red", "green", "blue"}.extract({true,false,true})
{"red", "blue"}
```

The second form takes an array of integers giving the indices to extract, as in the following example:

```
>> {"red","green","blue"}.extract({2,0,1,1})
{"blue", "red", "green", "green"}
```

You can create an empty array with a specific element type using the function `emptyArray`. For example, to create an empty array of integers, use:

```
>> emptyArray(int)
{}
```

You can combine arrays into a single array using the `concatenate` function. For example,

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

You can update an element of an array using the `update` function, for example,

```
>> {1, 2, 3}.update(0, 4)
{4, 2, 3}
```

The `update` function creates a new array[‡]

[‡]Actually, `update` creates a new token of type UpdatedArrayToken which keeps track of updated elements in a token while preserving the unchanged elements. The alternative would be to produce a new ArrayToken, which would result allocating memory and copying the entire source array.

13.3.2 Matrices

In Ptolemy II, arrays are ordered sets of tokens. Ptolemy II also supports **matrices**, which are more specialized than arrays. They contain only certain primitive types, currently *boolean*, *complex*, *double*, *fixedpoint*, *int*, and *long*. Currently *float*, *short* and *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, see Chapter 14).

Elements of matrices are referenced using *`matrixname`*`(`*`n`*`, `*`m`*`)`, where *`matrixname`* 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,

```
>> [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 element wise multiplication, use the `multipyElements` function (see Table 13.8). Matrix addition and subtraction are element wise, as expected, but the division operator is not supported. Element wise division can be accomplished with the `divideElements` function, and multiplication by a matrix inverse can be accomplished using the `inverse` function (see Table 13.8). A matrix can be raised to an *int*, *short* 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 13.7). 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
```

13.3.3 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 key of a field can be an arbitrary string, provided that it is quoted. Only strings that qualify as valid Java identifiers can be used without quotation marks. Note that quotation marks within a quoted string must be escaped using a backslash. 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).

An **ordered record** is similar to a normal record except that it preserves the original ordering of the labels. Ordered records are delimited using square brackets rather than curly braces. For example, `[b="foo", a=1]` is an ordered record token in which "b" will remain the first label.

Fields that are valid Java identifiers may be accessed using the period operator, optionally with braces—as if it were a method call. For example, the following two expressions:

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

both yield 1.

An alternative syntax to access fields uses the `get()` method. Note that this is the only way to access fields for which the key demands the use of quotation marks. For example:

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

yields 2.

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 for normal (unordered) records. Hence

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

Moreover, normal 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}
```

Equality comparisons for ordered records respect the original order of the fields. For example,

```
>> [a=1, b=2]==[b=2, a=1]
false
```

Additionally, ordered record fields are always reported in the order in which they are defined. For example,

```
>> [b=2, a=1]
[b=2, a=1]
```

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
```

Finally, You can create an empty record using the `emptyRecord` function:

```
>> emptyRecord()
{}
```

13.3.4 Union Types

Occasionally, more than one distinct data type will be sent over the same connection, or a variable may take on values with one of several data types. Ptolemy II provides a **union** type to accommodate this. A union type is designated as in the following example:

```
{|a = int, b = complex |}
```

This indicates a port or variable that may have type *int* or *complex*. A typical use of union types uses a **UnionMerge** and/or **UnionDisassembler** actor.

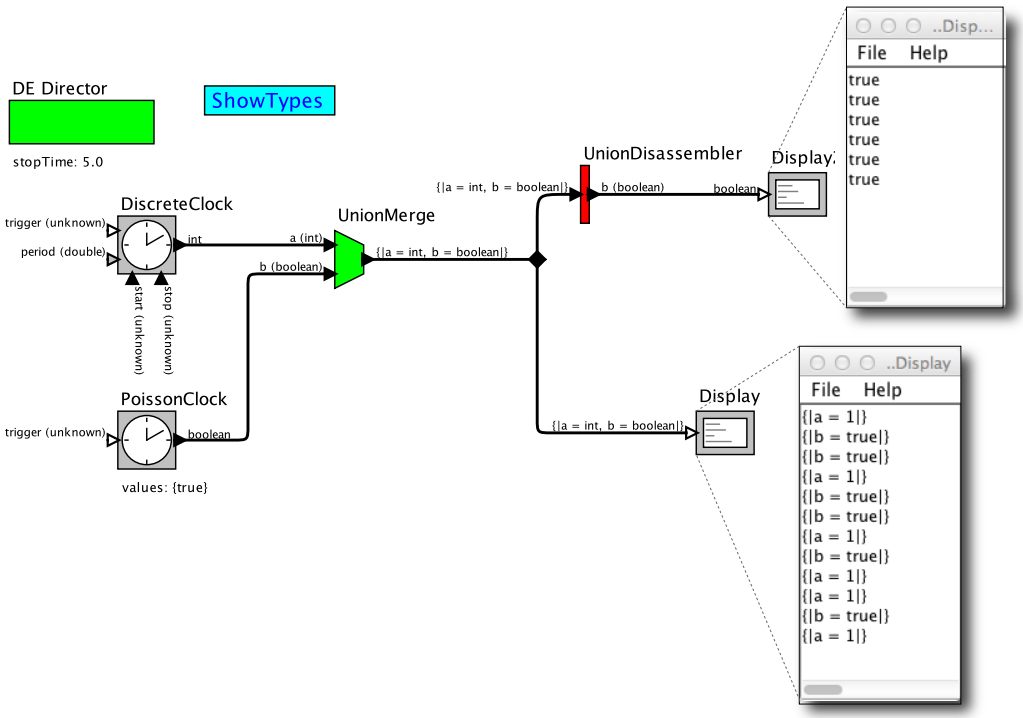


Figure 13.6: Union types allow types to resolve to more than one type. [online]

Example 13.1: Consider the example shown in Figure 13.6. This is a DE model with two data sources, a DiscreteClock, which produces outputs of type *int*, and a PoissonClock, which produces outputs of type *boolean*. These two streams of values are merged by the UnionMerge, whose output type becomes a union type. The names of the types in the union are determined by the names of the input ports of the UnionMerge, which are added when building the model. The lower Display actor displays the merged stream, showing that each token displayed has a single value, either an *int* or a *boolean*.

Along the upper path, a UnionDisassembler actor is used to extract from the stream the "b" types, which are *boolean*. Only those types are passed to the output, and the type of the output is inferred to be *boolean*.

13.4 Operations on Tokens

Every element and subexpression in an expression represents an instance of the Token class in Ptolemy II (or more likely, a class derived from Token). The expression language supports a number of operations on tokens that give access to the underlying Java code.

13.4.1 Invoking Methods

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()
```

13.4.2 Accessing Model Elements

Expressions in a model can reference elements of the model and invoke methods on them. The expression giving a parameter its value may reference by name any object that is contained by the container of the parameter.

Example 13.2: Figure 13.7 shows a model with four parameters *P1* through *P4*. The parameter *P1* has expression

```
Const2.value
```

The `Const2` here refers to the actor named "Const2" that is contained by the container of *P1*, and hence `Const2.value` refers to the *value* parameter of the actor Const2. The value of the parameter *P1* is therefore equal to the value of the *value* parameter of Const2.

The keyword `this` in a parameter expression refers to the object that contains the parameter.

Example 13.3: In Figure 13.7, Const2 has a *value* parameter with the expression (shown in its icon):

```
this.getName() + ": " + P2
```

Here, `this` refers to Const2, so `this.getName()` returns a string that is the name "Const2." The rest of the expression performs string concatenation, appending a colon and the value parameter *P2*.

The parameter *P2* has expression

```
this.entityList().size()
```

In this case, `this` refers to the container of *P2*, which is the top-level model. Hence, `this.entityList()` returns a list of entities (actors) contained by this top-level model. And finally, `this.entityList().size()` returns the number of actors contained by this top-level model, which is 5.

Now, the first two outputs of this model should be easy to understand:

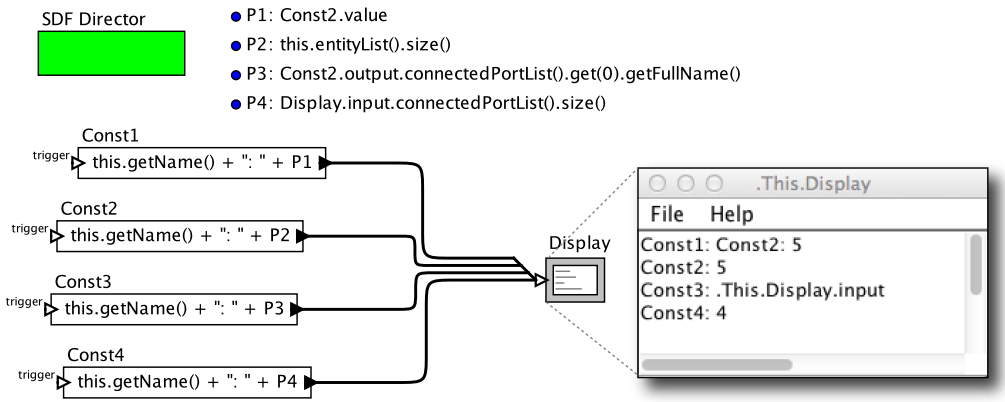


Figure 13.7: Expressions can access elements of the model, as shown here. [online]

```
Const1: Const2: 5
Const2: 5
```

The second output is the value of *P2* (namely 5) prepended by the name of the Const actor generating the output and a colon. The first output is the value *P1* (which is the string `"Const2: 5"`) prepended by the name of the Const actor generating the output and a colon.

Parameters can use method invocation to traverse the connections in the model, as illustrated next.

Example 13.4: In Figure 13.7, parameter *P3* has the expression:

```
Const2.output.connectedPortList().get(0)
          .getFullName()
```

This gets the full name of the first port in the list of ports that the *output* port of actor Const2 is connected to. Specifically, it returns the string `".This.Display.input"`, as displayed by actor Const3.

Similarly, *P4* has expression

```
Display.input.connectedPortList().size()
```

which returns the number of sources connected to the Display input.

For an overview of some of the methods that can be invoked on actors, parameters, and ports, see Chapter 12. For a complete listing, see the code documentation for Ptolemy II.

13.4.3 Casting

The cast function can be used to explicitly cast a value into a type. When the cast function is invoked with `cast(type, value)`, where `type` is the target type and `value` is the

value to be cast, a new value is returned (if a predefined casting is applicable) that is in the specified type. For example, `cast(long, 1)` yields 1L, which is equal to 1 but is in the long data type, and `cast(string, 1)` yields `"1"`, which is in the string data type.

13.4.4 Defining Functions

Users can define new functions in the expression language. 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
```

13.4.5 Higher-Order Functions

Functions can be passed as arguments to certain **higher-order functions** that have been defined (see Table 13.15). 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}
```

Ptolemy II also supports a fold function, which can be used to program a loop in an expression. The fold function applies a function to each element of an array, accumulating a result as it goes. The function that is folded over the array takes two arguments, the accumulated result so far and an array element. When the function to be folded is applied to the first element of the array, the accumulated result is an initial value.

Example 13.5:

```
fold(
    function(x:int, e:int) x + 1,
    0, {1, 2, 3}
)
```

This computes the length of array {1, 2, 3}. The result is 3, which is equal to {1, 2, 3}.length(). Specifically, the function to be folded is `function(x:int,`

`e:int) x + 1`. Given arguments `x` and `e`, it returns `x + 1`, ignoring the second argument `e`. It is first applied to the initial value, `0`, and the first element of the array, `1`, yielding `1`. It is then applied to the accumulated result, `1`, and the second element of the array, the value of which it ignores, yielding `2`. It is invoked the number of times equal to the number of elements in array {1, 2, 3}. Therefore, `x` is increased 3 times from the starting value 0.

Example 13.6: The following variant does not ignore the values of the array elements:

```
fold(
    function(x:int, e:int) x + e,
    0, {1, 2, 3}
)
```

This computes the sum of all elements in array {1, 2, 3}, yielding `6`.

Example 13.7:

```
fold(
    function(x:arrayType(int), e:int)
       e % 2 == 0 ?  x :  x.append({e}),
    {}, {1, 2, 3, 4, 5}
)
```

This computes a subarray of array {1, 2, 3, 4, 5} that contains only odd numbers. The result is {1, 3, 5}.

Example 13.8: Let C be an actor.

```
fold(
    function(list:arrayType(string),
       port:object("ptolemy.kernel.Port"))
```

```
        port.connectedPortList().isEmpty() ?
            list.append({port}) :  list,
    {}, C.portList()
)
```

This returns a list of `C`'s ports that are not connected to any other port (with `connectedPortList()` being empty). Each port in the returned list is encapsulated in an ObjectToken.

13.4.6 Using Functions in a Model

A typical use of functions in a Ptolemy II 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 Ptolemy II model.

Example 13.9: Consider the model shown in Figure 13.8. In that example, the Const actor defines a function that simply squares the argument. 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.

Example 13.10: A more elaborate use is shown in Figure 13.9 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 multiplies the output of the Ramp to the square of the output of the Ramp, thus computing the cube.

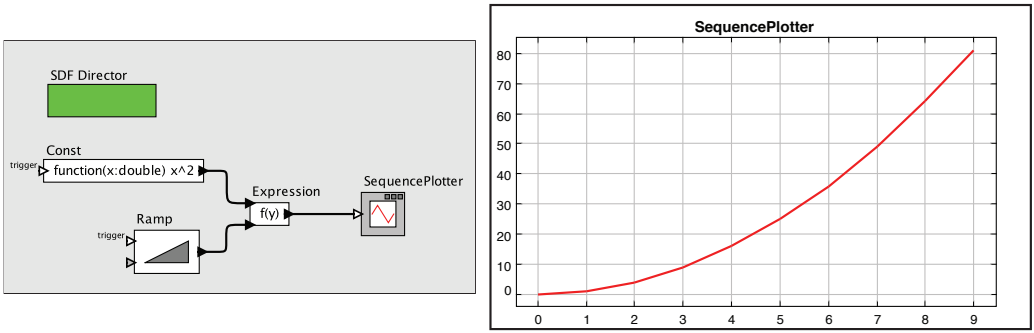


Figure 13.8: Example of a function being passed from one actor to another.

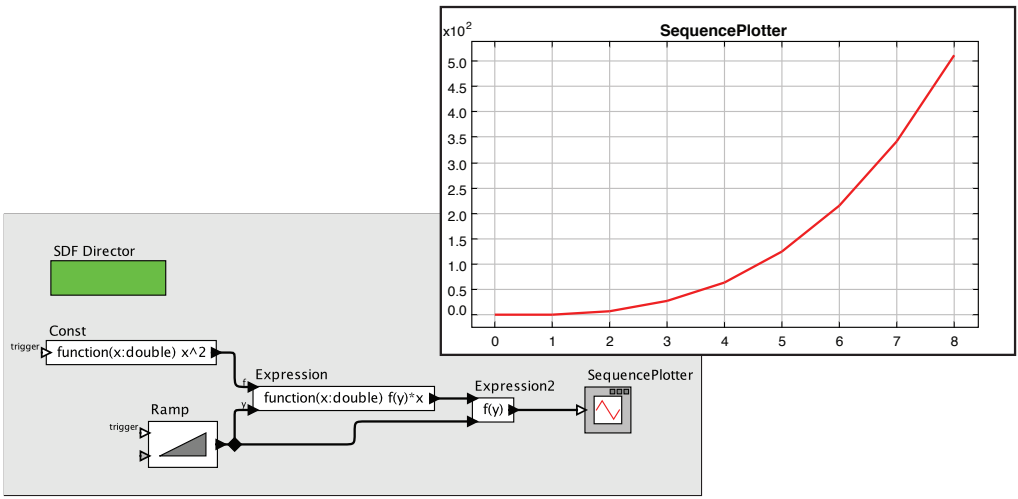


Figure 13.9: More elaborate example with functions passed between actors.

13.4.7 Recursive Functions

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:general, 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.

13.4.8 Built-In Functions

The expression language includes a set of functions, such as `sin`, `cos`, etc. The functions that are built in include all static methods [§] of the classes shown in Table 13.2, which together provide a rich set [¶].

The functions currently available are shown in the tables at the end of this chapter, which also show the argument types and return types. The argument and return types are the widest type that can be used. For example, `acos` will take any argument that can be losslessly cast to a *double*, such as unsigned byte, short, integer, float. *long* cannot be cast losslessly cast to *double*, so acos(1L) will fail. **Trigonometric functions** are given in Table 13.4. Basic **mathematical functions** are given in Tables 13.5 and 13.6. Functions that take or return matrices, arrays, or records are given in Tables 13.7 through 13.9. Utility functions for evaluating expressions are given in Table 13.10. Functions performing signal processing operations are given in Tables 13.11 through 13.13. I/O and other miscellaneous functions are given in Tables 13.15 and 13.16.

[§] Note that calling methods such as String.format() that have an argument of Object [] can be difficult because of problems specifying an array of Java Objects such as java.lang.Double instead of ptolemy.data.type.DoubleToken

[¶] Moreover, the set of available methods can easily be extended if you are writing Java code by registering another class that includes static methods (see the PtParser class in the ptolemy.data.expr package).

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*, *short*, 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 13.4 shows trigonometric functions. Note that these operate on *double* or *complex*, and hence on *int*, *short* and *unsignedByte*, which can be losslessly converted to *double*. The result will always be *double*. For example,

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

Table 13.2: The classes whose static methods are available as functions in the expression language.

java.lang.Math	ptolemy.math.IntegerMatrixMath
java.lang.Double	ptolemy.math.DoubleMatrixMath
java.lang.Integer	ptolemy.math.ComplexMatrixMath
java.lang.Long	ptolemy.math.LongMatrixMath
java.lang.String	ptolemy.math.IntegerArrayMath
ptolemy.data.MatrixToken.	ptolemy.math.DoubleArrayStat
ptolemy.data.RecordToken.	ptolemy.math.ComplexArrayMath
ptolemy.data.expr.UtilityFunctions	ptolemy.math.LongArrayMath
ptolemy.data.expr.FixPointFunctions	ptolemy.math.SignalProcessing
ptolemy.math.Complex	ptolemy.math.FixPoint
ptolemy.math.ExtendedMath	ptolemy.data.ObjectToken

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 13.7 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*, *short* 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 13.7 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 13.10 shows utility functions for evaluating expressions given as strings or representing numbers as strings. Of these, the `eval` function is the most flexible.

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 Ptolemy II was started, as reported by the Java Virtual Machine property `user.dir`), a user's home directory (as reported by the property `user.home`), or the classpath, which includes the directory tree in which Ptolemy II 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 `random` and `gaussian` functions shown in Table 13.5 and Table 13.6 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 Ptolemy II. 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 Java Virtual Machine system properties by name. Some possibly useful system properties are:

- ptolemy.ptII.dir: The directory in which Ptolemy II is installed.
- ptolemy.ptII.dirAsURL: The directory in which Ptolemy II 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.

For a complete list of Java Virtual Machine properties, see the Java documentation for `java.lang.System.getProperties`.

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 discrete cosine transform (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

$$X_k = s_k \sum_{n=0}^{N-1} x_n \cos((2n+1)k\frac{\pi}{2D}) \tag{13.1}$$

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 argument 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 13.3: Normalization options for the DCT function.

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

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

$$x_n = \sum_{k=0}^{N-1} s_k X_k \cos((2n+1)k\frac{\pi}{2D}). \tag{13.2}$$

13.5 Nil Tokens

Null or missing tokens are common in analytical systems like R and SAS where they are used to handle sparsely populated data sources. In database parlance, missing tokens are sometimes called null tokens. Since null is a Java keyword, we use the term "**nil**." Nil tokens are useful for analyzing real world data such as temperature where the value is not measured during every interval. In principle, one may want, for example, a TolerantAverage actor that does not require all data values to be present — when the TolerantAverage actor sees a nil token, it would ignore it. Note that this can lead to uncertainty. For example, if `average` is expecting 30 values and 29 of them are nil, then the average will not be very accurate.

In Ptolemy II, operations on tokens yield a nil token if any argument is a nil token. Thus, the Average actor is not like TolerantAverage. Upon receiving a nil token, all subsequent results will be nil. When an operation yields a nil value, the resulting nil token will have the same type that would normally have resulted from the operation, so type safety is preserved. Not all data types, however, support nil tokens. In particular, the various matrix types cannot have nil values because the underlying matrices are Java native type matrices that do not support nil.

The expression language defines a constant named `nil` that has value nil and type *niltype* (see Chapter 14). The `cast` expression language function can be used to generate nil values of other types. For example, "`cast(int, nil)`" will return a token with value nil and type *int*.

13.6 Fixed Point Numbers

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

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.

13.7 Units

Ptolemy II supports **units systems**, which are built on top of the expression language. Units systems allow parameter values to be expressed with units, such as "1.0 * cm", which is equal to "0.01 * meters". These are expressed this way (with the * for multiplication) because "cm" and "meters" are actually variables that become in scope when a units system icon is dragged in to a model. A few simple units systems are provided (mainly as examples) in the utilities library.

A model using one of the simple provided units systems is shown in Figure 13.10 This unit system is called *BasicUnits*; the units it defines can be examined by double clicking on its icon, or by invoking "Customize" |"Configure", as shown in Figure 13.11. In that figure, we see that "meters", "meter", and "m" are defined, and are all synonymous. Moreover, "cm" is defined, and given value "0.01*meters", and "in", "inch" and "inches" are defined, all with value "2.54*cm".

In the example in Figure 13.10, a constant with value "1.0 * meter" is fed into a Scale actor with scale factor equal to "2.0/ms". This produces a result with dimensions of length over time. If we feed this result directly into a Display actor, then it is displayed as "2000.0 meters/seconds", as shown in Figure 13.12, top display. The canonical units for length are meters, and for time are seconds.

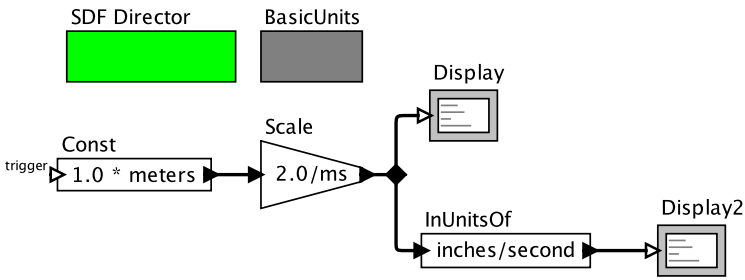


Figure 13.10: Example of a model that includes a unit system. [online]

In Figure 13.10, we also take the result and feed it to the InUnitsOf actor, which divides its input by its argument, and checks to make sure that the result is unitless. This tells us that 2 meters/ms is equal to about 78,740 inches/second.

The InUnitsOf actor can be used to ensure that numbers are interpreted correctly in a model, which can be effective in catching certain kinds of critical errors. For example, if in Figure 13.10, we had entered "seconds/inch" instead of "inches/second" in the InUnitsOf actor, we would have gotten the exception in Figure 13.13 instead of the execution in Figure 13.12.

Units systems are built entirely on the expression language infrastructure in Ptolemy II. The units system icons actually represent instances of scope-extending attributes, which are attributes whose parameters are in scope as if those parameters were directly contained by the container of the scope extending attribute. That is, scope-extending attributes can define a collection of variables and constants that can be manipulated as a unit. Two fairly extensive units systems are provided, CGSUnitBase and ElectronicUnitBase. Nonetheless, these are intended as examples only, and can no doubt be significantly improved and extended.

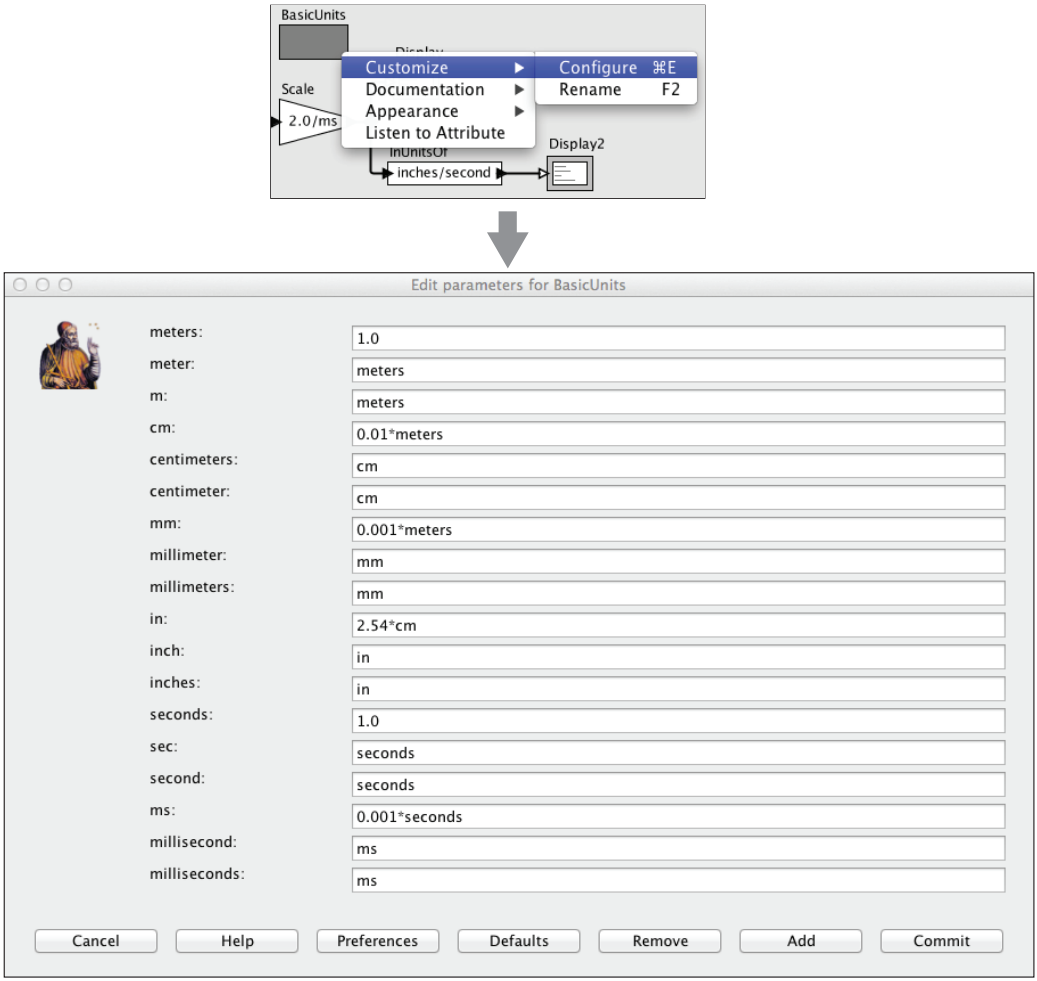


Figure 13.11: Units defined in a units system can be examined by double clicking or by right clicking and selecting Customize and Configure.

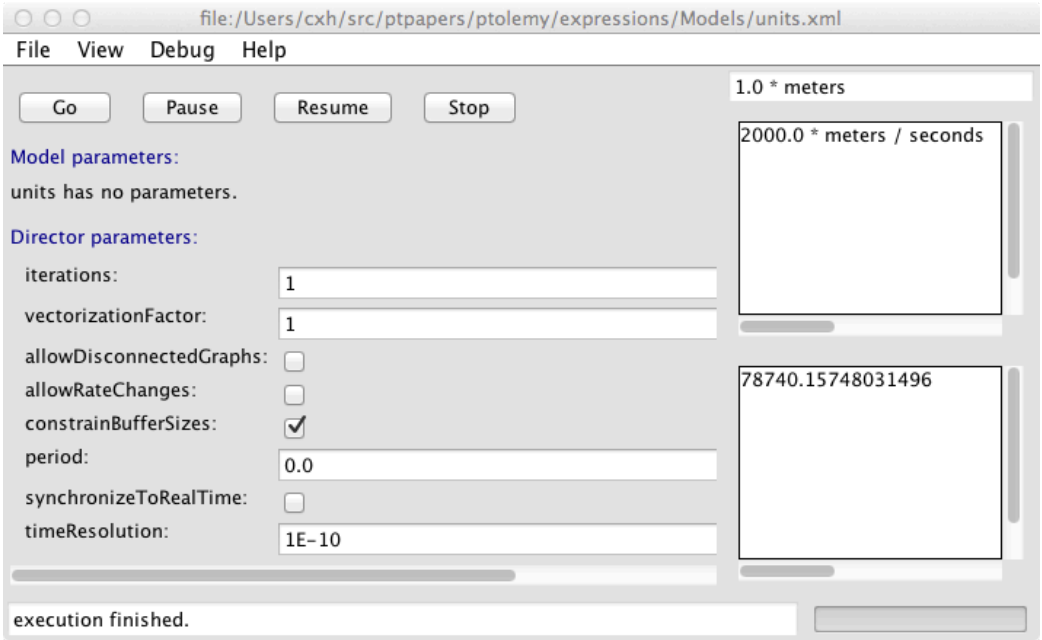


Figure 13.12: Result of running the model in Figure 13.10.

Figure 13.13: Example of an exception resulting from a units mismatch.

13.8 Tables of Functions

Table 13.4: Trigonometric functions.

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

Table 13.5: Basic mathematical functions, part 1.

Function	Argument type(s)	Return type	Description
abs	*double* or *complex*	*double* or *int* or *long*(complex returns *double*	absolute value *complex* case: $abs(a+ib) = \|z\| = \sqrt{a^2+b^2}$
angle	*complex*	*double* in the range $[-\pi, \pi]$	angle or argument of the *complex* number: $\angle z$
ceil	*double* or float	*double*	ceiling function, which returns the smallest (closest to negative infinity) *double* value that is not less than the argument and is an integer.
compare	*double*, *double*	*int*	compare two numbers, returning -1, 0, or 1 if the first argument is less than, equal to, or greater than the second.
conjugate	*complex*	*complex*	*complex* conjugate
exp	*double* or *complex*	*double* in the range [0.0, ∞] or complex	exponential function ($e^{argument}$) complex case: $e^{a+ib} = e^a(\cos(b) + i\sin(b))$
floor	*double*	*double*	floor function, which is the largest (closest to positive infinity) value not greater than the argument that is an integer.
gaussian	*double*, *double* or *double*, *double*, *int*, or *double*, *double*, *int*, *int*	*double* or *arrayType(double)* or [*double*]	one or more Gaussian random variables with the specified mean and standard deviation (see 13.4.8).
imag	*complex*	*double*	imaginary part
isInfinite	*double*	*boolean*	return true if the argument is infinite
isNaN	*double*	*boolean*	return true if the argument is "not a number"
log	*double* or *complex*	*double* or *complex*	natural logarithm *complex* case: $log(z) = log(abs(z) + i\angle z)$
log10	*double*	*double*	log base 10
log2	*double*	*double*	log base 2
max	*double*, *double* or double	a scalar of the same type as the arguments	maximum
min	*double*, *double* or double	a scalar of the same type as the arguments	minimum
pow	*double*, *double* or *complex*, *complex*	*double* or complex	first argument to the power of the second

Table 13.6: Basic mathematical functions, part 2.

Function	Argument type(s)	Return type	Description
random	no arguments or int or *int*, *int*	*double* or double or [*double*]	one or more random numbers between 0.0 and 1.0 (see 13.4.8)
real	*complex*	*double*	real part
remainder	*double*, *double*	*double*	remainder after division, according to the IEEE 754 floating-point standard (see 13.4.8).
round	*double*	*long*	round to the nearest long, 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	*double*	*int*	round to the nearest *int*, 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	*double*	*int*	-1 if the argument is negative, 1 otherwise
sqrt	*double* or complex	*double* or complex	square root. If the argument is *double* with value less than zero, then the result is NaN. complex case: $sqrt(z) = \sqrt{\|z\|}(\cos(\frac{\angle z}{2}) + i\sin(\frac{\angle z}{2}))$
toDegrees	*double*	*double*	convert radians to degrees
toRadians	*double*	*double*	convert degrees to radians
within	*type*, *type*, *double*	*boolean*	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.

Table 13.7: Functions that take or return matrices, arrays, or records, part 1.

Function	Argument type(s)	Return type	Description
arrayToMatrix	*arrayType(type)*, *int*, *int*	[*type*]	Create a matrix from the specified array with the specified number of rows and columns
concatenate	*arrayType(type)*, *arrayType(type)*	*arrayType(type)*	Concatenate two arrays.
concatenate	*arrayType(arrayType(type))*	*arrayType(type)*	Concatenate arrays in an array of arrays.
conjugateTranspose	[*complex*]	[*complex*]	Return the conjugate transpose of the specified matrix.
createSequence	*type*, *type*, *int*	*arrayType(type)*	Create an array with values starting with the first argument, incremented by the second argument, of length given by the third argument.
crop	[*int*], *int*, *int*, *int*, *int* or [*double*], *int*, *int*, *int*, *int* or [*complex*], *int*, *int*, *int*, *int* or [*long*], *int*, *int*, *int*, *int*	[*int*] or [*double*] or [*complex*] or [*long*] or	Given a matrix of any type, return a submatrix starting at the specified row and column with the specified number of rows and columns.
determinant	[*double*] or [*complex*]	*double* or complex	Return the determinant of the specified matrix.
diag	*arrayType(type)*	[*type*]	Return a diagonal matrix with the values along the diagonal given by the specified array.
divideElements	[*type*], [*type*]	[*type*]	Return the element-by-element division of two matrices
emptyArray	*type*	*arrayType(type)*	Return an empty array whose element type matches the specified token.
emptyRecord		*record*	Return an empty record.
find	*arrayType(type)*, *type*	*arrayType(int)*	Return an array of the indices where elements of the specified array match the specified token.
find	*arrayType(boolean)*	*arrayType(int)*	Return an array of the indices where elements of the specified array have value true.
hilbert	*int*	[*double*]	Return a square Hilbert matrix, where $A_{ij} = \frac{1}{i+j+1}$. A Hilbert matrix is nearly, but not quite singular.

Table 13.8: Functions that take or return matrices, arrays, or records, part 2.

Function	Argument type(s)	Return type	Description
identityMatrixComplex	*int*	[*complex*]	Return an identity matrix with the specified dimension.
identityMatrixDouble	*int*	[*double*]	Return an identity matrix with the specified dimension.
identityMatrixInt	*int*	[*int*]	Return an identity matrix with the specified dimension.
identityMatrixLong	*int*	[*long*]	Return an identity matrix with the specified dimension.
intersect	*record*, *record*	*record*	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	[*double*] or [*complex*]	[*double*] or [*complex*]	Return the inverse of the specified matrix, or throw an exception if it is singular.
matrixToArray	[*type*]	*arrayType(type)*	Create an array containing the values in the matrix
merge	*record*, *record*	*record*	Merge two records, giving priority to the first one when they have matching record labels.
multiplyElements	[*type*], [*type*]	[*type*]	Multiply element wise the two specified matrices.
orthonormalizeColumns	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal columns.
orthonormalizeRows	[*double*] or [*complex*]	[*double*] or [*complex*]	Return a similar matrix with orthonormal rows.
repeat	*int*, *type*	*arrayType(type)*	Create an array by repeating the specified token the specified number of times.
sort	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.
sortAscending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in ascending order. realScalar is any scalar token except *complex*.

Table 13.9: Functions that take or return matrices, arrays, or records, part 3.

Function	Argument type(s)	Return type	Description
sortDescending	*arrayType(string)* or *arrayType(realScalar)*	*arrayType(string)* or *arrayType(realScalar)*	Return the specified array, but sorted in descending order. realScalar is any scalar token except *complex*.
subarray	*arrayType(type)*, *int*, *int*	*arrayType(type)*	Extract a subarray starting at the specified index with the specified length.
sum	*arrayType(type)* or [*type*]	*type*	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	[*type*]	*type*	Return the trace of the specified matrix.
transpose	[*type*]	[*type*]	Return the transpose of the specified matrix.
update	*int*, *arrayType(type)*	*arrayType(type)*	Update an element in a an array.
zeroMatrixComplex	*int*, *int*	[*complex*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixDouble	*int*, *int*	[*double*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixInt	*int*, *int*	[*int*]	Return a zero matrix with the specified number of rows and columns.
zeroMatrixLong	*int*, *int*	[*long*]	Return a zero matrix with the specified number of rows and columns.

Table 13.10: Utility functions for evaluating expressions

Function	Argument type(s)	Return type	Description
eval	*string*	any type	evaluate the specified expression (see 13.4.8).
parseInt	*string* or *string*, *int*	*int*	return an *int* read from a string, using the given radix if a second argument is provided.
parseLong	*string* or *string*, *int*	*int*	return a long read from a string, using the given radix if a second argument is provided.
toBinaryString	*int* or *long*	*string*	return a binary representation of the argument
toOctalString	*int* or *long*	*string*	return an octal representation of the argument
toString	*double* or *int* or *int*, *int* or *long*	*string*	return a string representation of the argument, using the given radix if a second argument is provided.
traceEvaluation	*string*	*string*	evaluate the specified expression and report details on how it was evaluated (see 13.4.8).

Table 13.11: Functions performing signal processing operations, part 1.

Function	Argument type(s)	Return type	Description
close	*double*, *double*	*boolean*	Return true if the first argument is close to the second (within EPSILON, where EPSILON is a static public variable of this class).
convolve	*arrayType(double)*, *arrayType(double)* or *arrayType(complex)*, *arrayType(complex)*	*arrayType(double)* or *arrayType(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	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the Discrete Cosine Transform of the specified array, using the specified (optional) length and normalization strategy (see 13.4.8).
downsample	*arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(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	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, int *arrayType(complex)*, *int*	*arrayType(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.
generateBartlett Window	*int*	*arrayType(double)*	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: $w(n) = \begin{cases} 2\frac{n}{M}; & if 0 \leq n \leq \frac{M}{2} \\ 2 - 2\frac{n}{M}; & if \frac{M}{2} \leq n \leq M \end{cases}$

Table 13.12: Functions performing signal processing operations, part 2.

Function	Argument type(s)	Return type	Description
generateBlackman Window	*int*	*arrayType(double)*	Return a Blackman window with the specified length. For length M + 1, the formula is: $w(n) = 0.42 + 0.5\cos(\frac{2\pi n}{M}) + 0.08\cos(\frac{4\pi n}{M})$
generateBlackman HarrisWindow	*int*	*arrayType(double)*	Return a Blackman-Harris window with the specified length. For length M + 1, the formula is: $w(n) = 0.35875 + 0.48829\cos(\frac{2\pi n}{M}) + 0.14128\cos(\frac{4\pi n}{M}) + 0.01168\cos(\frac{6\pi n}{M})$
generateGaussian Curve	*arrayType(double)*, *arrayType(double)*, *int*	*arrayType(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).
generateHamming Window	*int*	*arrayType(double)*	Return a Hamming window with the specified length. For length M + 1, the formula is: $w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{M})$
generateHanning Window	*int*	*arrayType(double)*	Return a Hanning window with the specified length. For length M + 1, the formula is: $w(n) = 0.5 - 0.5\cos(\frac{2\pi n}{/}M)$
generatePolynomial Curve	*arrayType(double)*, *double*, *double*, *int*	*arrayType(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.

Table 13.13: Functions performing signal processing operations, part 3.

Function	Argument type(s)	Return type	Description
generateRaised CosinePulse	*double*, *double*, *int*	*arrayType(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: $h(t) = \frac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}} \times \frac{\cos(\frac{x\pi t}{T})}{1-(\frac{2xt}{T})^2}$, 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 excess Bandwidth of 0.0, this pulse is a sinc pulse.
generate Rectangular Window	*int*	*arrayType(double)*	Return an array filled with 1.0 of the specified length. This is a rectangular window.
IDCT	*arrayType(double)* or *arrayType(double)*, *int* or *arrayType(double)*, *int*, *int*	*arrayType(double)*	Return the inverse discrete cosine transform of the specified array, using the specified (optional) length and normalization strategy (see 13.4.8).
IFFT	*arrayType(double)* or *arrayType(complex)* or *arrayType(double)*, *int* *arrayType(complex)*, *int*	*arrayType(complex)*	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.

Table 13.14: Functions performing signal processing operations, part 4.

Function	Argument type(s)	Return type	Description
nextPowerOfTwo	*double*	*int*	Return the next power of two larger than or equal to the argument.
poleZeroTo Frequency	*arrayType(complex)*, *arrayType(complex)*, *complex*, *int*	*arrayType(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.
toDecibels	*double*	*double*	Return $20 \times \log_{10}(z)$, where z is the argument.
unwrap	*arrayType(double)*	*arrayType(double)*	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	*arrayType(double)*, *int*	*arrayType(double)*	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 $n > 0$.

Table 13.15: Miscellaneous functions, part 1.

Function	Argument type(s)	Return type	Description
asURL	*string*	*string*	Return a URL representation of the argument.
cast	type1, type2	type1	Return the second argument converted to the type of the first, or throw an exception if the conversion is invalid.
constants	none	*record*	Return a record identifying all the globally defined constants in the expression language.
findFile	*string*	*string*	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.
filter	function, *arrayType(type)*	*arrayType(type)*	Extract a sub-array consisting of all of the elements of an array for which the given predicate function returns true.
filter	function, *arrayType(type)*, *int*	*arrayType(type)*	Extract a sub-array with a limited size consisting of all of the elements of an array for which the given predicate function returns true.
freeMemory	none	*long*	Return the approximate number of bytes available for future memory allocation.
iterate	function, *int*, *type*	*arrayType(type)*	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	function, *arrayType(type)*	*arrayType(type)*	Return an array that results from applying the specified function to the elements of the specified array.
property	*string*	*string*	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, ptolemy.ptII.dir, ptolemy.ptII.dirAsURL, and user.dir.

Table 13.16: Miscellaneous functions, part 2.

Function	Argument type(s)	Return type	Description
readFile	*string*	*string*	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. The readfile function is often used with the eval function.
readResource	*string*	*string*	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	*long*	Return the approximate number of bytes used by current objects plus those available for future object allocation.
yesNoQuestion	*string*	*boolean*	Query the user for a yes-no answer and return a boolean. This function will open a dialog if a GUI is available, and otherwise will use standard input and output.

14 The Type System

Edward A. Lee, Marten Lohstroh, and Yuhong Xiong

Contents

In a programming language, a **type system** associates a type with each variable. A **type** is logically a family of values that the variable can take on. For example, the type *double* is the set of all double-precision floating point numbers represented by 64 bits according to the IEEE 754 standard for floating-point arithmetic. A **strong type system** will prevent a program from using the 64-bit value of a *double* variable as, for example, a pointer into memory or a *long* integer (Liskov and Zilles, 1974). Java has a strong type system; C does not. A good introduction to type systems is given by Cardelli and Wegner (1985).

The Ptolemy II type system associates a type with each port and parameter in a model. Ptolemy II uses **type inference**, where the types of parameters and ports are inferred based on their usage. Types need not be declared by the model builder, usually.

A programming language where types are checked at compile time is said to be **statically typed**. In Ptolemy II, types are checked just prior to execution of a model, between the preinitialize and initialize phases of execution. Since this happens once, before execution of the model, we consider Ptolemy II to be statically typed.

Although the type system is a strong one (a port will not receive a token that is incompatible with its declared type, for example), there are loopholes. In particular, users can define their own actors in Java, and these actors may not behave well. For example, an actor may declare an output port to be of type *int* and then attempt to send a *string* through that port. To catch such errors, Ptolemy II also performs run-time type checks. Although it is rare (unless you write your own actors), it is possible to build models that will exhibit type errors only during execution. These errors will not be detected by the static type checker.

Fortunately, with the help of static type checking, run-time type checks can be performed automatically when a token is sent out from a port. The run-time type checker simply compares the type of a produced token against the (static) type of the output port. This way, a type error is detected at the earliest possible time (when the token is produced, rather than when it is used). However, this does not guarantee that a type of token that an actor accepts is indeed compatible with the operation it implements. A run-time type error may therefore also be thrown by the actor itself, particularly if the actor is written incorrectly.

The Ptolemy II type system supports **polymorphism**, where actors can operate on a variety of data types. To facilitate the construction of polymorphic actors, the type system offers a mechanism called **automatic type conversion**, which allows a component to receive multiple data types by automatically converting them to a single data type, assuming

that the conversion can be done without loss of information. Polymorphism greatly increases the reusability of actors in the presence of static typing, especially in combination with **type inference**. In this chapter we describe how these mechanisms are integrated into the Ptolemy II static type checking framework, with emphasis on how they help build correct models.

14.1 Type Inference, Conversion, and Conflict

In Ptolemy II models, types of ports are inferred from the model, subject to constraints imposed by the actors and the infrastructure. We will explain exactly how these constraints come about and how the inference is performed, but first we can develop an intuitive understanding of what happens.

Example 14.1: Consider the example shown in Figure 14.1. This model calculates and displays $1 - \pi$. (This is a silly model, since the entire model could be replaced by the expression `1 - PI`, but it serves to illustrate the type system.) The model includes an attribute called *ShowTypes*, which can be found in the `Utilities→Analysis` library. This attribute causes Vergil to display the type of each port next to the port (in addition to the name of the port, if the name would otherwise be displayed). As you can see in the figure, initially the type of each port is *unknown*.

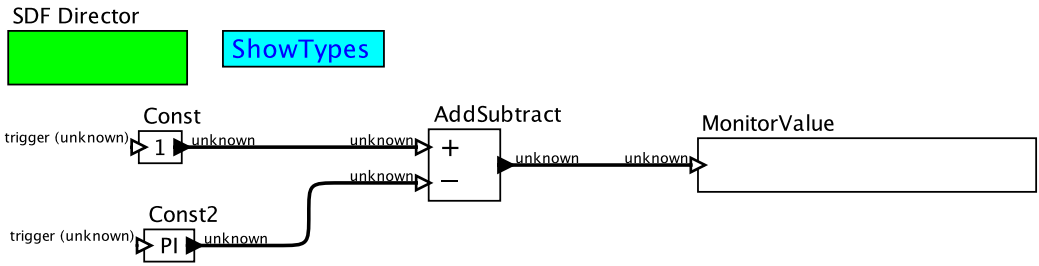


Figure 14.1: A simple example for illustrating type inference and conversion. [online]

During the first stage of execution, preinitialize, the types of the ports are inferred, after which the display is eventually updated as shown in Figure 14.2. The types of the output ports of the Const actors are determined by their *value* parameters, which are `1` and `PI`, respectively. If you change a *value* parameter to, for example, `1+i`, then the output type will become *complex*. If you change it to `{1, 2}`, then the output type will become *arrayType(int, 2)*, an array with *int* elements and length 2.

Notice in Figure 14.2 that the input ports of the AddSubtract actor have both resolved to *double*. The AddSubtract actor imposes a constraint that its two input ports must have the same type. When the AddSubtract actor receives an *int* token from the Const actor, the input port will automatically convert the token to type *double*. We will explain in detail what conversions are allowed, but intuitively, a conversion is allowed if no information is lost.

Notice in Figure 14.2 that the output port of the AddSubtract actor and the input port of MonitorValue have also resolved to *double*. The MonitorValue actor can accept any input type, since it simply displays a string representation of the token, and every token has a string representation.

The previous example illustrates that the type of a parameter in one part of a model can have far-reaching consequences in other parts of the model. The type system ensures consistency. It is common when building models to raise type errors, as illustrated by the following example.

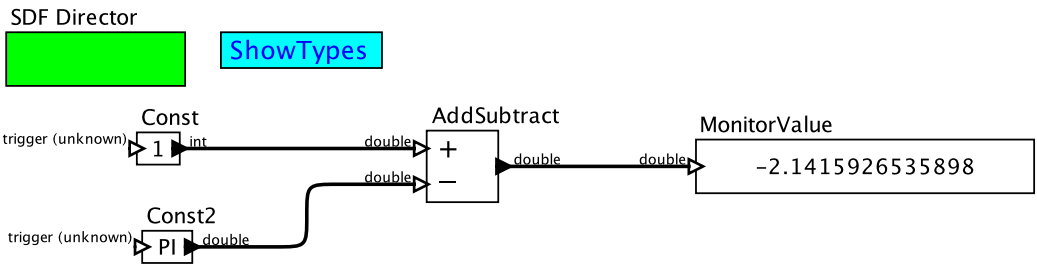


Figure 14.2: After execution.

Example 14.2: Consider the example shown in Figure 14.3. In that example, we have replaced MonitorValue with SequencePlotter, and we have changed the Const actor to produce a *complex* value. Type inference determines that the output of AddSubtract is *complex*. But SequencePlotter requires an input of type *double*. A *complex* token cannot be losslessly converted to a *double* token, so upon executing the model you will get the following exception:

```
ptolemy.actor.TypeConflictException: Type conflicts occurred on
the following inequalities:
  (port .TypeConflict.AddSubtract.output: complex) <=
  (port .TypeConflict.SequencePlotter.input: double)
  in .TypeConflict
```

This error message reports that a type constraint in the model cannot be satisfied. That type constraint is that the type of the output port of AddSubtract must be *less than or equal to* ($\leq$) the type of the input port of SequencePlotter. Further, it reports that the output port has type *complex*, while the input port has type *double*. The offending ports and their containers are then highlighted in the model as shown in the figure.

In the above example, a type constraint is given as an inequality, an assertion that the type of one port must be "less than or equal to" the type of another. What does this mean?

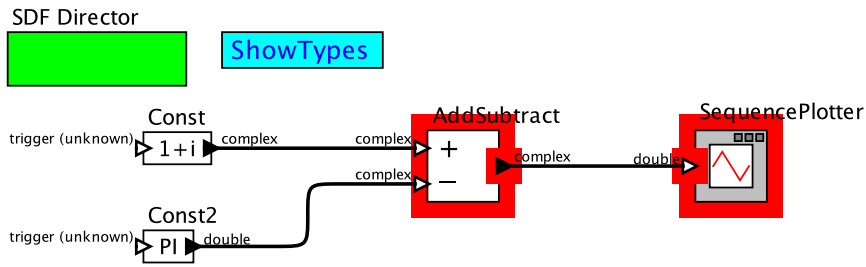


Figure 14.3: Type conflict.

Intuitively, one type is less than another if it can be lossless converted to that other type. We examine this inequality relation next.

14.1.1 Automatic Type Conversion

The allowed automatic type conversions are represented in Figure 14.4, which depicts the **type lattice** of Ptolemy II. In this diagram, a conversion from a first type to a second type is allowed if there is an upward path from the first type to the second type in the diagram. This relationship implies a partial order on types (see sidebar on page 513), so we might say that a conversion is allowed if the first type is less than or equal to the second type. This partial order has an elegant mathematical structure called a lattice (see sidebar on page 513) that enables efficient type inference and type checking.

Automatic conversions occur when an actor retrieves data from its input port.* The types of ports are determined prior to execution, and run-time type checking guarantees that tokens sent through an output port are compatible with the types of downstream input ports (i.e., a conversion to such type is allowed by the type lattice). This is due to the type constraints that are imposed by connections between ports. These type constraints are explained in Section 14.1.2. If a token is not compatible, the run-time type checker will throw an exception before the token is sent. Hence, run-time type errors are detected as early as possible.

A type conversion can also be forced in the expression language using the `cast` function, one of many built-in functions available in the expression language. An expression of the form `cast(newType, value)` will convert the specified value into the specified type. See Section 13.4.3 and Table 13.15 for information about the `cast` function.

The type lattice is constructed based on a principle of lossless conversion. A conversion is allowed automatically as long as important information about value of data tokens is not lost. Such conversions are referred to as **widening conversions** in Java. For instance, converting a 32-bit signed integer to a 64-bit IEEE double precision floating point number is allowed, since every integer can be represented exactly as a floating point number. On the other hand, data type conversions that lose information are not automatic.

* Some actors disable automatic type conversion on their input ports because they do not need the conversion. For example, AddSubtract and Display accept any token type, because they make use of methods that are inherited by all token types. These actors disable automatic conversion by invoking: `portName.setAutomaticTypeConversion(false)`.

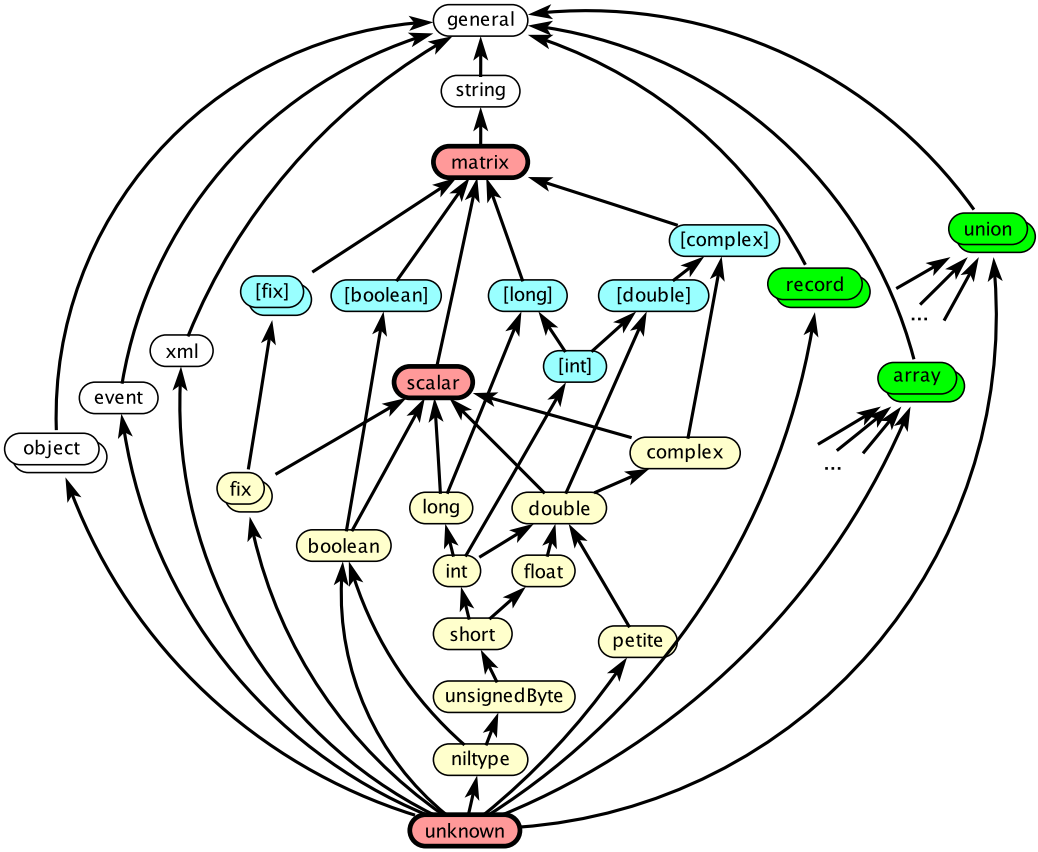


Figure 14.4: The Type Lattice. Types that cannot be instantiated are shown with bold outlines (and pink fill). Scalar types are lightly filled (in yellow), matrix types are slightly more darkly filled (in cyan), and composite types still more darkly filled (in green). The composite types and the *fix* types are infinite sublattices, as suggested by their double icons.

Sidebar: What is a Lattice?

A lattice is a mathematical structure that has properties that enable efficiently solving type constraints. A lattice is a set with particular kind of **order relation** related to CPOs (see sidebar on page 182). See Davey and Priestly (2002) for more details.

First, a total order is an ordering over a set S, denoted $\leq$, where any two elements of the set are ordered. Specifically, for any $x, y \in S$, either $x \leq y$ or $y \leq x$ (or both, in which case $x = y$). For example, if the set S is the set of integers, and $\leq$ denotes the ordinary arithmetic ordering, then $(S, \leq)$ is a total order.

A partial order relaxes the constraint that any two elements be ordered. An example of a partial order is $(S, \leq)$, where S is a set of sets and $\leq$ is the subset relation, usually denoted $\subseteq$. Specifically, if A, B are both sets in S, then it may be that neither $A \subseteq B$ nor $B \subseteq A$. E.g., let $A = \{1, 2\}$ and $B = \{2, 3\}$; then neither is a subset of the other.

Another partial order is the **prefix order** on strings. Let S be the set of sequences of alphabetic characters, for example. Then for two strings $x, y \in S$, we write $x \leq y$ is x is a prefix of y. E.g., if $x =$abc and $y =$abcd, then $x \leq y$. If $z =$bc, then neither $x \leq z$ nor $z \leq x$ holds. Formally, a partial order is a set S and a relation $\leq$, such that for all $x, y, z \in S$,

- $x \leq x$ (the order is reflexive),
- if $x \leq y$ and $y \leq z$, then $x \leq z$ (the order is transitive), and
- if $x \leq y$ and $y \leq x$, then $x = y$ (the order is antisymmetric).

The least upper bound (LUB), if it exists, of a subset $U \subseteq S$ of a partial order $(S, \leq)$ is the least element $x \in S$ such that for every $u \in U$, $u \leq x$. The **greatest lower bound** (**GLB**) of U, if it exists, is the greatest element $x \in S$ such that for every $u \in U$, $x \leq u$. E.g., in the prefix order, if $x =$abc, $y =$abcd, and $z =$bc, then the LUB of $\{x, y\}$ is y. The LUB of $\{x, z\}$ does not exist. The GLB of $\{x, y\}$ is x. The GLB of $\{x, z\}$ is the empty string, which is a prefix of all strings.

A **lattice** is a partial order $(S, \leq)$ for which every subset of S has a GLB and a LUB. The subset order is a lattice, because the LUB can be found with set union, and the GLB can be found with set intersection. The prefix order on strings, however, is not a lattice, because two strings may not have a LUB. The prefix order is a **lower semi-lattice**, however, because the GLB of a set of strings always exists.

14.1.2 Type Constraints

A model imposes a number of constraints that drive type inference. A constraint is expressed as an inequality between the types of two ports. It requires one port to have a type that is less than or equal to (losslessly convertible to) the type of the other port.

In a Ptolemy II topology, the **type compatibility rule** requires an output port to have a type that is less than or equal to all inputs to which it is connected, as follows:

$$outType \leq inType \tag{14.1}$$

This constraint guarantees that there is an allowed automatic conversion that can be performed during data transfer. Every connection between an output port and an input port establishes a type constraint that enforces this rule.

Example 14.3: In Figure 14.2, the Const actor produces type *int*, while the AddSubtract actor receives type *double*. In Figure 14.4, we see that *int* is less than *double*, so the type compatibility rule is satisfied.

In addition to the constraints imposed by the connections between actors, most actors also impose constraints. For example, the AddSubtract actor declares that its output type is greater than or equal to its input types, and that the types of its two input ports are equal. An equality constraint is equivalent to two inequality constraints, as in:

$$\begin{aligned} plus &\leq minus \\ minus &\leq plus, \end{aligned}$$

where *plus* and *minus* are the input ports of the AddSubtract actor.

Many actors impose a **default type constraint** that requires an unconstrained input to be less than or equal every unconstrained output. By default, actors implicitly include this type constraint for every set of input and output ports that have no explicit type constraint.

Example 14.4: Some actors operate on tokens without regard for the actual types of the tokens. For example, the DownSample does not care about the type of token

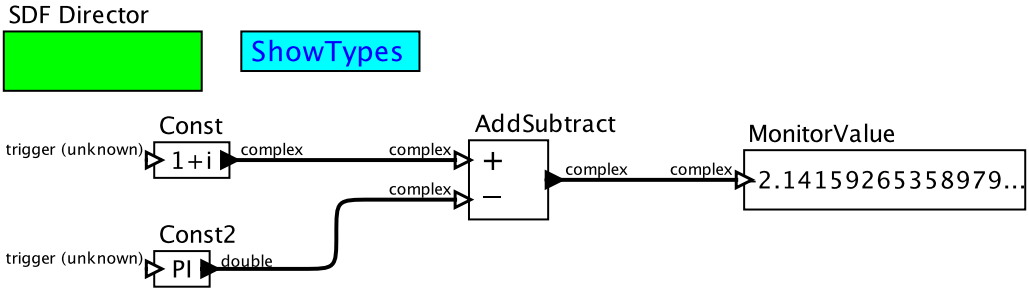


Figure 14.5: Type conflict of Figure 14.3 resolved by using an actor that can accept any input type, MonitorValue. [online]

going through it, so it does not explicitly declare any type constraints. The default type constraint enables type information to propagate through this actor from its input to its output.

By default, actors that leave their input ports undeclared will have the type of the input port determined by the upstream model (unless the model has enabled backward type inference, as explained below in Section 14.1.4).

Example 14.5: The MonitorValue actor, which displays the value of tokens it receives, can accept any type of input. By default, it leaves its input undeclared, which results in the type resolving to whatever is provided upstream. For example, Figure 14.5 resolves the type conflict of Figure 14.3 by replacing the SequencePlotter with a MonitorValue actor. The resolved type of the input, *complex*, is determined by the upstream actors.

14.1.3 Type Declarations

Sometimes, there is not enough information in a model to infer types from the sources of data.

Example 14.6: Consider for example the model in Figure 14.6. This model is intended to evaluate expressions entered by a user in a shell, but type resolution fails, as shown. The ExpressionToToken actor takes an input string, which is expected to be an expression in the Ptolemy expression language (see Chapter 13), and evaluates the expression. The result of evaluation is produced on the output. There is no way to anticipate what the user might type in the shell, so there is not enough information to infer types. The type of the output of the ExpressionToToken remains *unkonwn*.

Such difficulties can be fixed by enabling backward type inference (discussed below), or by explicitly declaring the type of a port, as illustrated in the next example.

Example 14.7: We can force the type of output of the ExpressionToToken actor to be of type *general*, as shown in Figure 14.7. Downstream types resolve to *general* as well.

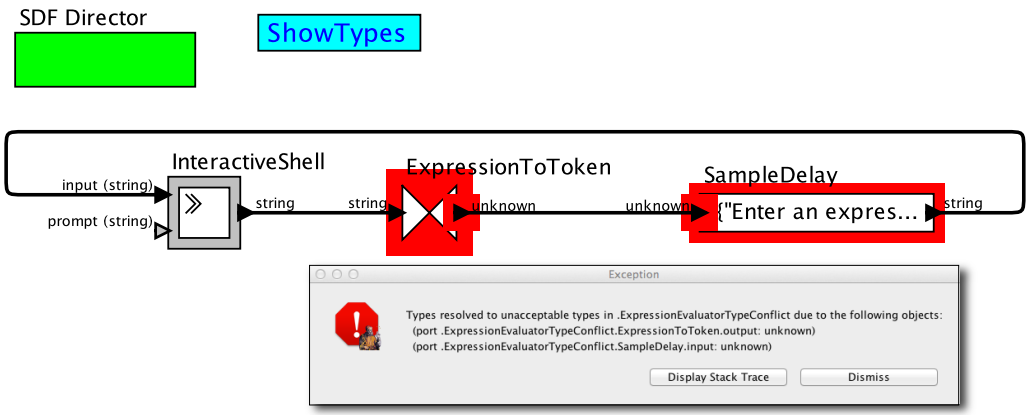


Figure 14.6: A model intended to evaluate expressions entered by a user, but for which there is not enough information for types to be inferred from the sources of data.

In the type column of the port dialog, you can enter any expression in the expression language. Whatever type that expression resolves to will be the declared type of the corresponding port. For clarity, Ptolemy II provides some built in variables that designate a type. The variable named `general`, for instance, evaluates to a token of type *general*. Similarly, the variable named `double` evaluates to a token of type *double* (which happens to have value 0.0, but the value immaterial). Table 14.8 lists the predefined variable names and their corresponding types.

14.1.4 Backward Type Inference

In all the examples discussed so far, type inference propagates forward in models, with each constant or fixed output type causing downstream types to resolve. That is, type information travels in the same direction that the tokens are sent during execution. The type compatibility rule given by (14.1) imposes no useful constraints on output ports, because it is always satisfied if *outType* has type *unknown*, the bottom element of the type lattice. Nevertheless, **forward type inference** is usually sufficient because sources of data in most models provide specific type information about those data.

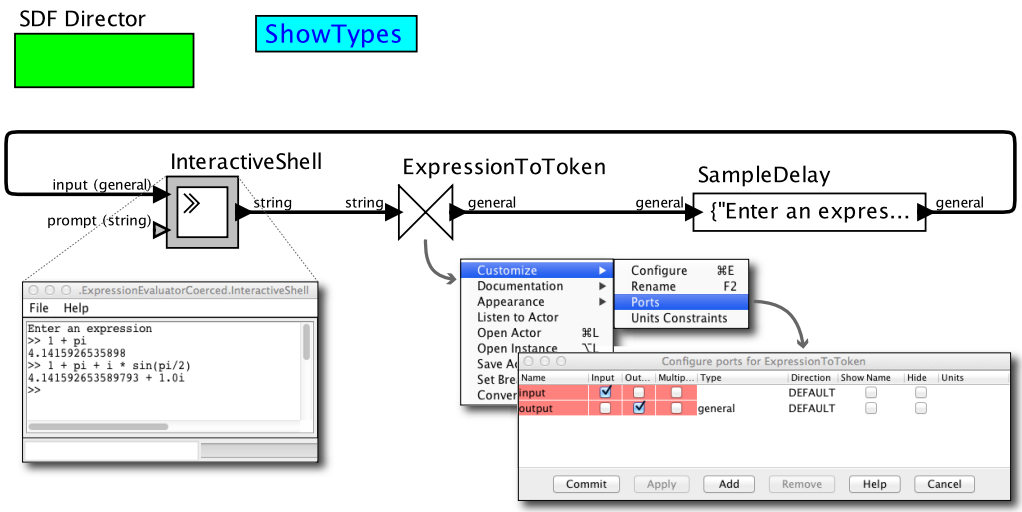


Figure 14.7: Types of ports declared by entering a type into the type column of the port configuration dialog. [online]

BaseType field	Expression	Description
UNKNOWN	`unknown`	bottom element of the data type lattice
ARRAY_BOTTOM		*array* of unknown type
BOOLEAN	`boolean`	boolean (true or false)
BOOLEAN_MATRIX	`[boolean]`	matrix of booleans
UNSIGNED_BYTE	`unsignedByte`	unsigned byte
COMPLEX	`complex`	complex number
COMPLEX_MATRIX	`[complex]`	complex matrix
FLOAT	`float`	32-bit IEEE floating-point number
DOUBLE	`double`	64-bit IEEE floating-point number
DOUBLE_MATRIX	`[double]`	matrix of doubles
FIX	`fixedpoint`	fixed-point data type
FIX_MATRIX	`[fixedpoint]`	matrix of fixed-point numbers
SHORT	`short`	16-bit integer
INT	`int`	32-bit integer
INT_MATRIX	`[int]`	matrix of 32-bit integers
LONG	`long`	64-bit integer
LONG_MATRIX	`[long]`	matrix of 64-bit integers
OBJECT	`object`	object type
ACTOR		actor type
XMLTOKEN		XML type
SCALAR	`scalar`	scalar number
MATRIX		matrix of unknown type
STRING	`string`	string
GENERAL	`general`	any type
EVENT		event (empty token)
PETITE		a double constrained to be between -1 and 1
NIL	`nil`	nil type
RECORD		record type

Figure 14.8: Type constants defined in the BaseType class with their corresponding name in the expression language, if there is one.

The models in Figures 14.6 and 14.7, however, do not have this property. First, since the model forms a loop, there is no clear "source" of data. Every actor is both upstream and downstream of every other actor. Moreover, the ExpressionToToken actor, by nature of what it does, cannot provide any specific information about the type of its output.

The Ptolemy II type system optionally provides **backward type inference** to solve this problem. To enable backward type inference, set the *enableBackwardTypeInference* parameter to true at the top level of the model, as shown in Figure 14.9. This has three effects. First, it causes certain actors that do not impose restrictions on the data received at input ports to declare those ports to have type *general*. This includes InteractiveShell and Display, for example. Second, it allows type constraints to propagate upstream. Specifically, it adds an additional constraint to the type compatibility rule of (14.1). The additional constraint is that the type of each output port is required to be greater than or equal to the greatest lower bound (GLB) of the types of all input ports to which it is connected. Third, for each actor that does not explicitly constrain the type relationships of its port, it imposes a default type constraint that the types of its input ports are greater than or equal to the GLB of the types of its output ports. These additional constraints are sufficient for the types to resolve to the same solution that we achieved with type coercion in Figure 14.7.

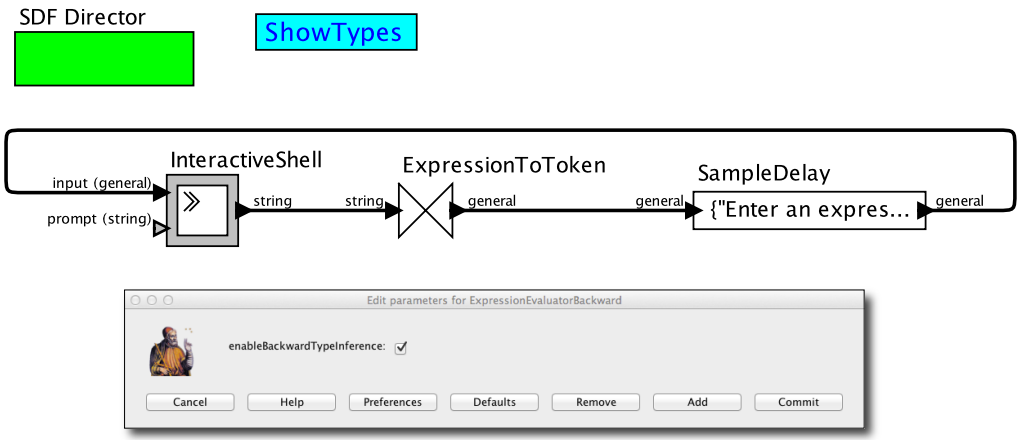


Figure 14.9: Enabling backward type inference allows type constraints to propagate upstream. [online]

The root of the problem is the ExpressionToToken actor, which cannot itself be specific about its output type. Any constraints on its output type would have to result from how its output tokens are used, rather than from the actor itself. In Figure 14.9, the InteractiveShell input accepts type *general*, which therefore propagates upstream to the output of the ExpressionToToken actor. When there are undeclared output port types, such as on the ExpressionToToken actor, backward type inference finds the the most general type that is compatible with downstream type constraints.

14.2 Structured Types

14.2.1 Arrays

Structured types include those tokens which aggregate other tokens of arbitrary type, such as array and record types. As described in Section 13.3, an array is an ordered list of tokens, all of which have the same type. Records contain a set of labeled tokens, like a struct in the C language. It is useful for grouping multiple pieces of related information together. In the type lattice in Figure 14.4, record types are incomparable with all the base types except *unknown*, *string*, and *general*. Array types are a bit more complex because any type is less than an array of that type in the type lattice. This is hinted at in the figure with the disconnected lines at the bottom of the array type. Note that the lattice nodes *array* and *record* actually represent an infinite number of types, so the type lattice is infinite.

For any type a, the following type relation holds,

$$a < \{a\}.$$

A value can be losslessly converted to an array of values. Moreover,

$$a < b \quad \Rightarrow \quad \{a\} < \{b\}.$$

Combining these, we see that the definition is recursive, so

$$a < \{a\} < \{\{a\}\} < \{\{\{a\}\}\} \cdots$$

Example 14.8: $int \leq double$, so the following all hold:

$$\begin{aligned} int &< \{int\} \\ \{int\} &< \{double\} \\ int &< \{double\} \\ int &< \{\{double\}\} \\ &\ldots \end{aligned}$$

A consequence of these type relations is that there is an infinite path from any particular array type to the top of the type lattice. This can result in situations where type inference does not converge.

Example 14.9: In the model in Figure 14.10, the Expression actor constructs an array consisting of one element, its input token. When you attempt to run this model, an exception occurs that includes the message

```
Large type structure detected during type resolution
```

The reason for this is that there is no (finite) type that satisfies all the constraints. The SampleDelay actor requires its output to be at least an *int*, because it produces an initial *int*. But it also requires that its output be greater than equal to its input. The first input it will receive will have type $\{int\}$, and the second input will have type $\{\{int\}\}$, etc. The only possible type is an infinite nesting of arrays.

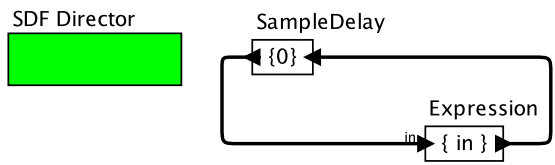


Figure 14.10: Example where type inference does not converge.

The Ptolemy II type system often (but not always) includes the *length* of an array in its type. If you explicitly query the type of a token, you can see this, as in the following command in the expression evaluator:

```
>> {1, 2}.getType()
object(arrayType(int,2))
```

Thus, `arrayType(int,2)` is the type of an array of length two, whereas `arrayType(int)` is the type of an array with indeterminate length. Including the length in the type allows the type system to detect more errors than otherwise.

Generally speaking, array types with specific length are incomparable with array types with different lengths, and can be converted to an array type with unknown length (and compatible element type). Scalars are convertible to array types with length 1.

One subtlety is that when you specify an array type with an expression `{ int }`, you are actually giving the type `arrayType(int, 1)`, which is more specific than you probably want. For this reason, unless you specifically want to constrain array types, it is better to specify an array type with `arrayType(int)`.

14.2.2 Records

The order relation between two record types follows the standard **depth subtyping** and **width subtyping** relations commonly used for such types (Cardelli, 1997). In depth subtyping, a record type c is a subtype of a record type d (i.e., $c \leq d$) if the fields of c are a subtype of the corresponding fields in d. For example,

```
{x = string, y = int} <= {x = string, y = double}
```

In width subtyping, a record with more fields is a subtype of a record with fewer fields. For example, we have:

```
{x = string, y = double, z = int} <= {x = string, y = double}
```

The width subtyping rule is a bit counterintuitive, as it implies a type conversion which loses information, discarding the extra fields of a record. However, it conforms with the "**is a**" interpretation of types, where $a \leq b$ if a is a b. In this case, the record with more fields "is an" instance of the record with fewer fields, whereas the reverse is not true.

14.2.3 Unions

Another structured type is the union type. It allows the user to create a token that can hold data of various types, but only one at a time. This is like the union construct in C. The union type is also called a **variant type** in the type system literature. The width subtyping relation for union type is the opposite to that of the record type. That is, a shorter union is a subtype of a longer one. Again, this corresponds with the "is a" interpretation of type relations.

A consequence of this width subtyping relation is that there are an infinite number of types from a particular union type to the top of the type lattice. This again means that type inference may not converge. The Ptolemy II type system truncates type inference after a finite number of steps, providing a heuristic that is a likely indicator of a type error.

14.2.4 Functions

One final structured type is the expression language function, described in Section 13.4.4. Functions can take several arguments and return a single value. The type system supports **function types**, where the arguments have declared types, and the return type is known. Function types are related in a way that is contravariant (oppositely related) between inputs and outputs. Namely, if `function(x:int, y:int) int` is a function with two integer arguments that returns an integer, then

```
function(x:int, y:int) int <= function(x:int, y:int) double
function(x:int, y:double) int <= function(x:int, y:int) int
```

The contravariant notion here is easiest to think about in terms of the automatic type conversion of one function into another. A function that returns *int* can be converted into a function that returns *double* by adding a conversion of the returned value from *int* to *double*. On the other hand, a function that takes an *int* cannot be converted into a function that takes a *double*, since that would mean that the function is suddenly able to accept *double* arguments when it could not before, and there is no automatic conversion from *double* to *int*. Functions that are lower in the type lattice assume less about their inputs and guarantee more about their outputs.

The names of arguments do not affect the relation between two function types, since argument binding is by the order of arguments only. Additionally, functions with different numbers of arguments (different **arity**) are considered incomparable.

The presence of function types that can be used as any other token results in what is commonly termed a higher-order type system. An example of the use of function tokens is given in Figure 2.44 and discussed in the sidebar on page 89.

14.3 Type Constraints in Actor Definitions

Section 12.4 introduces how to write actors in Java. In this section, we explain how to constrain types in the Java definition of an actor.

Prior to the execution of a model, during the setup phase, type constraints are gathered from all entities in the model (e.g., instances of TypedIOPort, TypedAtomicActor, or Parameter) that impose restrictions on their types. Actors can either set type constraints by storing them in the object instances of the concerning ports or parameters, or by setting them up using the `_customTypeConstraints` method of TypedAtomicActor.

A simple type constraint that is common to many actors is to ensure that the type of an output is greater than or equal to the type of a parameter. You can do so by putting the following statement in the constructor:

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

or equivalently, by defining:

```
protected Set<Inequality> _customTypeConstraints() {
  Set<Inequality> result = new HashSet<Inequality>();
  result.add(new Inequality(parameterName.getTypeTerm(),
    portName.getTypeTerm()));
  return result;
}
```

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

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

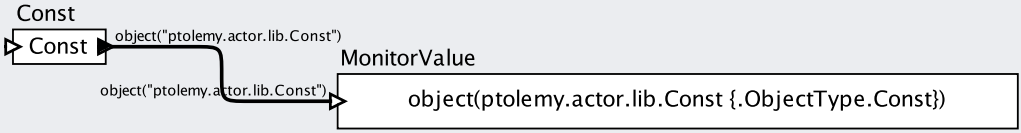
These constraints could be specified in reverse order,

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

which obviously means the same thing.

Sidebar: Object Types

The *object* type at the far left in Figure 14.4 is particularly powerful (and should be used with caution). A token of type *object* represents a Java object, such as a Ptolemy actor. Consider the following model:



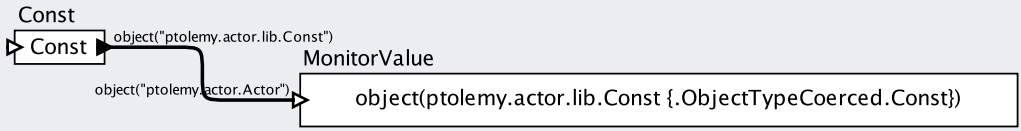
The *value* of the Const actor is set to `Const`, which evaluates to the Const actor itself. The Const actor's name is "`Const`," and the expression `Const` evaluates to the actor itself. Hence, the inferred type of the output port of Const is *object*, and its output will be the actor itself.

The *object* type is not one type, but an infinite number of types, as suggested by the double oval in Figure 14.4. There is a particular *object* type for every distinct Java class. The type of the output port above is `object("ptolemy.actor.lib.Const")`, because the Const actor is an instance of the Java class `ptolemy.actor.lib.Const`.

A type `object("A")` is less than `object("B")` if the Java class `A` is a subclass of Java class `B`. For example, `ptolemy.actor.lib.Const` implements the Java interface, `ptolemy.actor.Actor`, so

```
object("ptolemy.actor.lib.Const") <= object("ptolemy.actor.Actor")
```

In the following variant of the above model, the input port of the MonitorValue actor is coerced to `object("ptolemy.actor.Actor")`:



A similar conversion can be accomplished with the `cast` function by doing

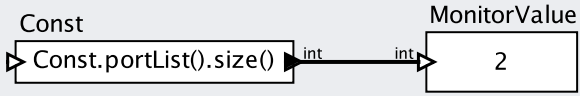
```
cast(object("ptolemy.actor.Actor"), Const)
```

The most general object type is *object* (without any argument). The pre-defined object token called `null` has this type.

Sidebar: Invoking Methods on Object Tokens

The expression language permits methods defined in the Java class of an object token (see sidebar on page 525). For example, in a model that contains an actor named C, the term `C` in an expression may refer to that actor.

Java methods may be invoked on the objects encapsulated in object tokens. For example, in the following model, the Const actor outputs the number of ports contained by the Const actor:



Sidebar: Petite and Unsigned Byte Data Types

The **petite** data type is used to represent real numbers in the range between -1.0 and 1.0 inclusive. It is used to emulate the behavior in certain specialized processors such as DSP processors (Digital Signal Processing), which sometimes use fixed-point arithmetic limited to this range of values. The *petite* type approximates this as a *double* limited to the range between -1.0 and 1.0 inclusive. In the expression language, a *petite* number is indicated by the suffix "p", and arithmetic operations saturate at -1.0 and 1 when results would lie outside this range. For example, using the expression evaluator, we get

```
>> 0.5p + 1.0p
1.0p
```

A data type that is sometimes useful for operating on raw data (e.g. packets arriving from a network) is the **unsigned byte**, designated as follows:

```
>> 1ub
1ub
>> -1ub
255ub
```

The same constraints can be expressed like this:

```
protected Set<Inequality> _customTypeConstraints() {
  result.add(new Inequality(parameterName.getTypeTerm(),
      portName.getTypeTerm());
  result.add(new Inequality(portName.getTypeTerm(),
      parameterName.getTypeTerm());
  return result;
}
```

The `_customTypeConstraints` method is particularly useful for actors that establish type constraints between ports that may be dynamically removed or added. For those actors, it is not safe to store the type constraints in the respective ports because constraints associated with no longer existing ports can persist and inadvertently cause type errors.

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

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

The above line declares that the port can only handle doubles. Figure 14.8 lists the type constants defined in the BaseType class. Another form of absolute type constraint imposes an upper bound on the type:

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

which declares that any type that can be losslessly converted to ComplexToken is acceptable. By default, for any input port that has no declared type constraints, a default type constraint is automatically created that declares its type to be less than or equal to that of any output ports that have no declared type constraints. If there are input ports with no constraints, but no output ports lacking constraints, then those input ports will remain unconstrained. Conversely, if there are output ports with no constraints, but no input ports lacking constraints, then those output ports will remain be unconstrained. The latter is unacceptable, unless backward type inference is enabled. Default type constraints can be disabled by overriding the `_defaultTypeConstraints` method and having it return null.

A port can be declared to accept any token using following type constraint:

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

Example 14.10: An extension of Transformer of Figure 12.11 is shown in Figure 14.11. This SimplerScale is a simplified version of the Scale actor in the standard Ptolemy II library. This actor produces an output token on each firing with a value that is equal to a scaled version of the input. The actor is polymorphic in that it can support any token type that supports multiplication by the factor parameter. In the constructor, the output type is constrained to be at least as general as both the input and the factor parameter.

```
1  public class SimplerScale extends Transformer {
2       public SimplerScale(CompositeEntity container,
3                 String name)
4                 throws NameDuplicationException,
5                 IllegalActionException {
6            super(container, name);
7            factor = new Parameter(this, "factor");
8            factor.setExpression("1");
9            // set the type constraints.
10           output.setTypeAtLeast(input);
11           output.setTypeAtLeast(factor);
12      }
13      public Parameter factor;
14      public Object clone(Workspace workspace)
15                throws CloneNotSupportedException {
16           SimplerScale newObject = (SimplerScale)super.
17                     clone(workspace);
18           newObject.output.setTypeAtLeast(newObject.input);
19           newObject.output.setTypeAtLeast(newObject.factor);
20           return newObject;
21      }
22      public void fire() throws IllegalActionException {
23           if (input.hasToken(0)) {
24                Token in = input.get(0);
25                Token factorToken = factor.getToken();
26                Token result = factorToken.multiply(in);
27                output.send(0, result);
28           }
29      }
30 }
```

Figure 14.11: Actor with non-trivial type constraints.

Notice in Figure 14.11 how the `fire` method uses `hasToken` to ensure that no output is produced if there is no input. Furthermore, only one token is consumed from each input channel, even if there is more than one token available. This is generally the behavior of domain polymorphic actors. Notice also how it uses the `multiply` method of the `Token` class. This method is polymorphic. Thus, this scale actor can operate on any token type that supports multiplication, including all the numeric types and matrices.

An awkward complication when customizing type constraints is illustrated by the `clone` method in lines 12-18. In order for the actor to work properly in actor-oriented classes, relative type constraints that are set up in the constructor have to be repeated in the `clone` method.

The `setTypeAtLeast`, `setTypeAtMost`, `setTypeEquals`, and `setTypeSameAs` methods are part of the Typeable interface, which is implemented by ports and parameters. The `setTypeAtMost` method is usually invoked on input ports to declare a requirement that *input* tokens must satisfy, while the `setTypeAtLeast` method is usually invoked on *output* ports to declare a guarantee of the type of the output. The methods `_customTypeConstraints` and `_defaultTypeConstraints` are part of the base class TypedAtomicActor, and its subclasses can override those methods to customize the type constraints they impose.

Example 14.11: The constraint that the type of an input port can be no greater than double might be declared as:

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

Note that the argument to `setTypeAtMost` and `setTypeEquals` is a type, whereas the argument to `setTypeAtLeast` is a Typeable object. This reflects the common usage, where `setTypeAtLeast` is declaring a dependency on externally provided types, whereas both `setTypeAtMost` and `setTypeEquals` declare constraints on externally defined types. The forms of the type inequalities that are specifiable by these methods also ensures that type inference is efficient and that the result of type inference is deterministic.

More complex type constraints arise from structured types, such as arrays and records. To declare that a parameter is an array of doubles, use:

```
parameter.setTypeEquals(new ArrayType(BaseType.DOUBLE));
```

This declares that a parameter or a port has a particular array type. A more flexible parameter might be able to contain an array of any type. This is expresses as follows:

```
parameter.setTypeAtLeast(ArrayType.ARRAY_BOTTOM);
```

In a more elaborate example, we might constrain the type of an output port to be no less than the element type of the array contained by a parameter (or an input port):

```
outputPort.setTypeAtLeast(ArrayType.arrayOf(parameter));
```

To declare that an output has a type greater than or equal to that of the elements of an input (or parameter) array, use:

```
outputPort.setTypeAtLeast(ArrayType.elementType(inputPort));
```

The above code implicitly constrains the input port to have an array type, but does not constrain the element types of that array. The above kinds of constraints appear in source actors such as DiscreteClock and Pulse, ArrayToSequence and SequenceToArray. Examining the source code for those actors can be instructive.

Another common constraint is that an input port of an actor receives a record with unconstrained fields. This constraint can be declared using the following code:

```
inputPort.setTypeAtMost(RecordType.EMPTY_RECORD);
```

Suppose you have an output port that may produce records with arbitrary fields. The above construct will not be sufficient since it does not declare any lower bound on the type, so at run time, the type will not be resolved to something useful. Instead, do:

```
outputPort.setTypeEquals(BaseType.RECORD);
```

This forces the type to resolve to the empty record. Any record with fields is a subtype of the empty record type, so this effectively declares the output to produce any record. Alternatively, enabling backward type inference allows the element type of the record to be inferred from the type constraints imposed by downstream actors.

To declare that a parameter can have a value that is any record, you can do:

```
param.setTypeAtMost(BaseType.RECORD);
```

but you will also need to specify a value for the parameter so that the type resolves to something concrete. To give a default value that is an empty record you can do:

```
param.setToken(RecordToken.EMPTY_RECORD);
```

Two of the types, *matrix* and *scalar*, are union types. This means that an instance of this type can be any of the types immediately below them in the lattice. An actor may, for example, declare that an input port must be of type no greater than scalar:

```
inputPort.setTypeAtMost(BaseType.SCALAR);
```

In this case, inputs of any type immediately below scalar in the type lattice will not be converted, except that the type of the input tokens will be reported as scalar. This is useful, for example, in actors that need to compare tokens, such as the Limiter actor. The `fire` method of that actor contains the code

```
if (input.hasToken(0)) {
    ScalarToken in = (ScalarToken) input.get(0);
    if ((in.isLessThan((ScalarToken) bottom.getToken()))
            .booleanValue()) {
        output.send(0, bottom.getToken());
    } else if ((in.isGreaterThan((ScalarToken) top.getToken()))
            .booleanValue()) {
        output.send(0, top.getToken());
    } else {
        output.send(0, in);
    }
}
```

This code relies on input port *in* and parameter *bottom* being declared to be at most scalar type, and ScalarToken being a base class for every token with type immediately below scalar. It then uses comparison methods defined in the ScalarToken class.

Type constraints in actors can get much more sophisticated than what we describe here. As always, the source code (and its extensive documentation) is the ultimate reference.

14.4 Summary

Ptolemy II includes a sophisticated type system that performs inference and checks for errors. It uses an efficient algorithm given by Rehof and Mogensen (1999), who prove that their algorithm has complexity that is linear in the number of occurrences of symbols in the type constraints. As a consequence, the algorithm scales well to large models. Most

of the time, the type system makes it unnecessary for the builder of models to think about types. At it makes it easy to define actors that operate on a multiplicity of types, as most of the actors in the standard library do.

Sidebar: Monotonic Functions in Type Constraints

More sophisticated type constraints can be expressed using a **monotonic function** on the left-hand side of an inequality. A monotonic function f preserves the order of its arguments; that is

$$x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2). \tag{14.2}$$

Using a monotonic function, it is possible to define a type that is dependent on other types in complicated ways. For example, the actor RecordDisassembler sets up a type constraint that forces each field of its input record to be of the same type as the output port with the same name as the field.

A monotonic function is specified by subclassing the abstract class MonotonicFunction and implementing the methods `getVariables` and `getValue`. The `getVariables` method returns the variables that the function takes as arguments. The `getValue` method returns the result of applying the function to the current value of the variables it depends on.

For example, the ConstructAssociativeType class subclasses MonotonicFunction. The variables returned by `getVariables` are the variables holding the types of a list of ports, such as the output ports of a RecordDisassembler actor. The `getValue` method returns a record type with fields matching the names of those ports and types matching the types of those ports.

It should be noted that the base class MonotonicFunction does not guarantee that its subclasses indeed behave monotonically. If a function that is not actually monotonic is used in a type constraint, then type resolution is no longer guaranteed to yield a unique result. The result may depend on the order in which constraints are applied.

15 Ontologies

Patricia Derler, Elizabeth A. Latronico, Edward A. Lee, Man-Kit Leung, Ben Lickly, Charles Shelton

Contents

An **ontology** in information science refers to an explicit organization of knowledge. An ontology can be organized into a graph as a set of **concepts** and the relations between those concepts. By constructing an ontology over a specific domain, a user is formalizing information of that domain in a way that can be shared with others. Models can have annotations added to them that express how they are used with respect to an ontology. Ontology-based annotations are a form of model documentation. Like type signatures, they can express the intended use of a model, but with respect to the domain of the ontology rather than to the type system.

A **static analysis** of a program or model is a check that can be run at compile time. Ptolemy II's type checker (see Chapter 14) is one example of a static analysis. It infers the data types used throughout a model and checks for consistency. In fact, the Ptolemy II type system is an ontology. It is an organization of knowledge about the data that a model operates on. The ontology checker described in this chapter also performs inference based on the annotations, and then checks consistency. But it is not constrained to checking data types. Instead, ontologies can be used to express static analyses from a variety of user-defined domains, including, for example:

- **units checking**: determining whether the units of data are consistent;
- **constant analysis**: determining what data in a model is constant, and what data varies in time;
- **taint analysis**: determining whether values in a data stream are influenced by an untrusted source; and
- **semantics checking**: determining whether the meaning of data produced by one component is consistent with the meaning assumed by another component that uses the data.

Such analyses can expose a variety of modeling errors.

Example 15.1: A portion of a model of a multi-tank fuel system in an aircraft (Moir and Seabridge, 2008) is shown in Figure 15.1. This model has three actors, where the ports are labeled with names that suggest the intended meaning and units of the data that are exchanged between actors. The model has three types of errors. It has units errors, where for example one component gives the level of a tank in gallons to a component that assumes that the level is being given in kilograms. (The latter is often a better choice, since amount of fuel in gallons varies with temperature, whereas the amount in kilograms does not.) It also has semantics

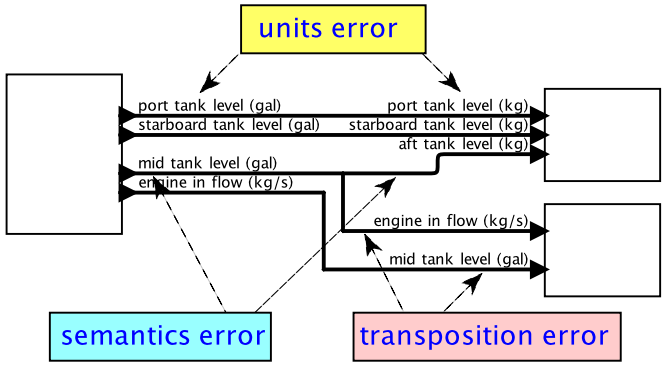


Figure 15.1: An illustration of some of the sorts of errors that can be caught by an ontology system.

errors, where a component gives the level of mid tank to a component that assumes it is seeing the level of the aft tank. And it has a transposition error, where a level and a flow are exchanged.

Such modeling errors are extremely easy to make and can have devastating consequences. This chapter gives an overview of how to construct ontologies and use them to prevent such errors.

15.1 Creating and Using Ontologies

The ontologies package provides an analysis that can be run on top of an existing model, so the first step is to create a Ptolemy II model on which we can run our analysis. In this section, we use a rather trivial model and a rather trivial ontology to illustrate the mechanics of construction of an ontology and the use of a solver. We will then illustrate a less trivial ontology that is practical and useful for catching certain kinds of modeling errors.

Example 15.2: Figure 15.2, shows a simple model with both constant and non-constant actors. The constant actors produce a sequence of output values that are all the same. For this model, we will show how to create a simple analysis that checks which signals in the model are constant. To do this, we will first define an ontology that distinguishes the concept of "constant" from "non constant." We will then define constraints for actors used by the model, and finally, we will invoke the solver.

Sidebar: Background on the Ontology Framework

The approach to ontologies described in this chapter was first given by Leung et al. (2009). They build on the theory of Hindley-Milner type systems (Milner, 1978), the efficient inference algorithm of Rehof and Mogensen Rehof and Mogensen (1996), the implementation of this algorithm in Ptolemy II (Xiong, 2002), and the application of similar mathematical foundations to formal concept analysis (Ganter and Wille, 1998).

An interesting extension of this basic mechanism, devised by Feng (2009), uses ontologies to guide model-based **model transformation**, where a Ptolemy II model modifies the structure of another Ptolemy II model. For example, the constant analysis described in Section 15.1 can guide a model optimization that replaces all constant subsystems with a Const actor. Also, Lickly et al. (2011) show how an infinite lattice can be used to not just infer that a signal is constant, but also to infer its value. Lickly et al. (2011) also show various other ways to use infinite lattices, including unit systems. They also show how ontologies work with structured types such as records.

The **Web Ontology Language** (**OWL**) is a widely-used family of languages endorsed by the World Wide Web Consortium (**W3C**) for specifying ontologies. OWL ontologies, like ours, form a partial order with a top and bottom element, but unlike ours, they are not constrained to be a lattice. Hence, the efficient inference algorithm of Rehof and Mogensen cannot always be applied. Nevertheless, a very useful extension of the mechanisms described in this chapter would be to export and import OWL ontologies. The **Eclipse Modeling Framework** (**EMF**) also specifies ontologies through the notion of classes and subclasses. Many Eclipse-based tools have been developed supporting it, so again it would be useful to build bridges.

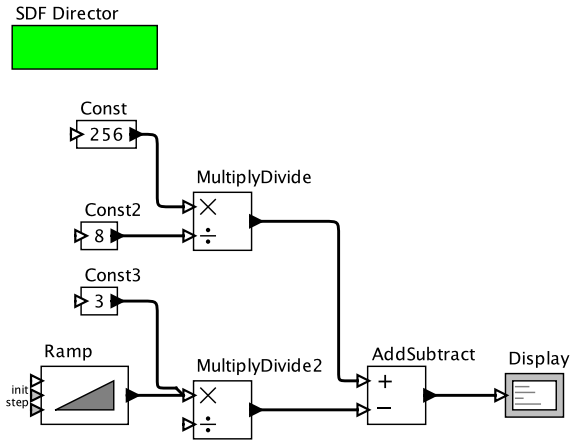


Figure 15.2: A simple Ptolemy model made of constant and non-constant actors and containing an ontology solver that can determine which signals are constant.

15.1.1 Creating an Ontology

In order to create the analysis, the first step will be to add the solver that will perform our analysis. In this case, we will drag in the **LatticeOntologySolver** actor to our model, as shown in Figure 15.3. This is where we will add all of the details of how our analysis works. These include the lattice that represents the concepts that we are interested in, and the constraints that actors impose on those concepts. In our case, the lattice will specify whether a signal is constant or not, and the constraints will provide information about which actors produce constant or non-constant signals.

As shown in Figure 15.3, if you open the LatticeOntologySolver, you get an editor with a customized library for building analyses. At a minimum, an analysis requires an ontology. Figure 15.4 illustrates the steps in constructing one. First, drag into the blank editor an **Ontology**. Open the ontology and drag **Concept**s into it from the library provided by the ontology editor.

First, we should assign meaningful names to our concepts. In Figure 15.5, we have renamed Top to NonConstant, Bottom to Unused, and Concept to Constant. We have also edited the parameters to the NonConstant concept to change its color to a light red, and to check the *isAcceptable* parameter, which visually removes the bold outline. These con-

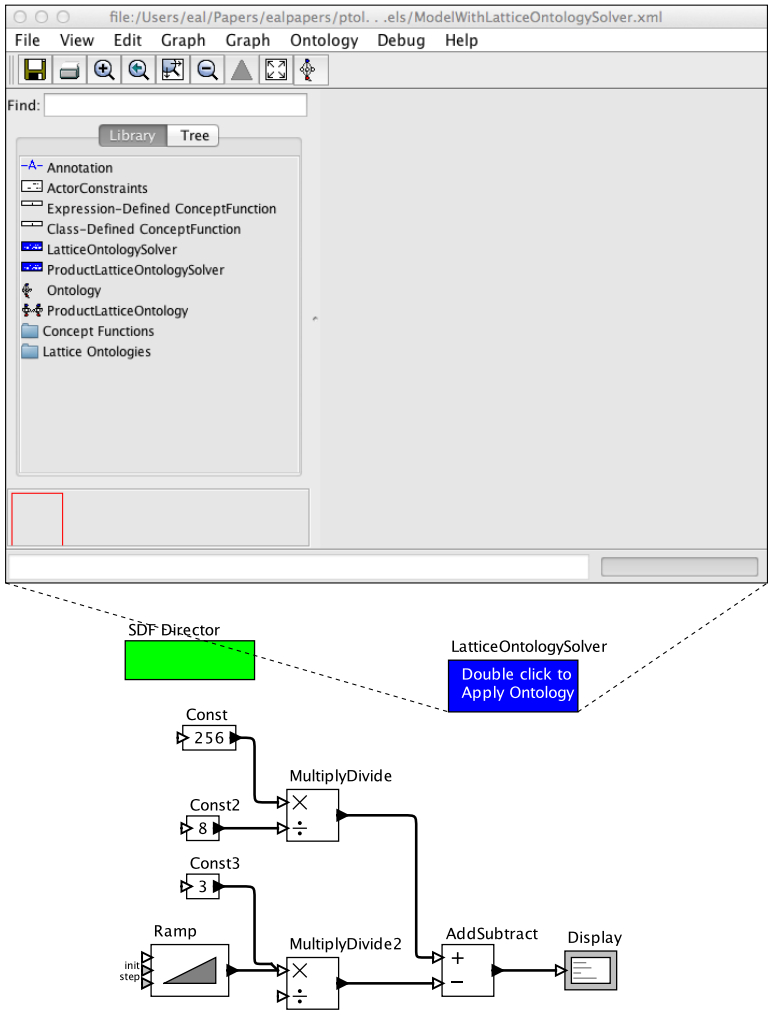


Figure 15.3: A model with a blank ontology solver.

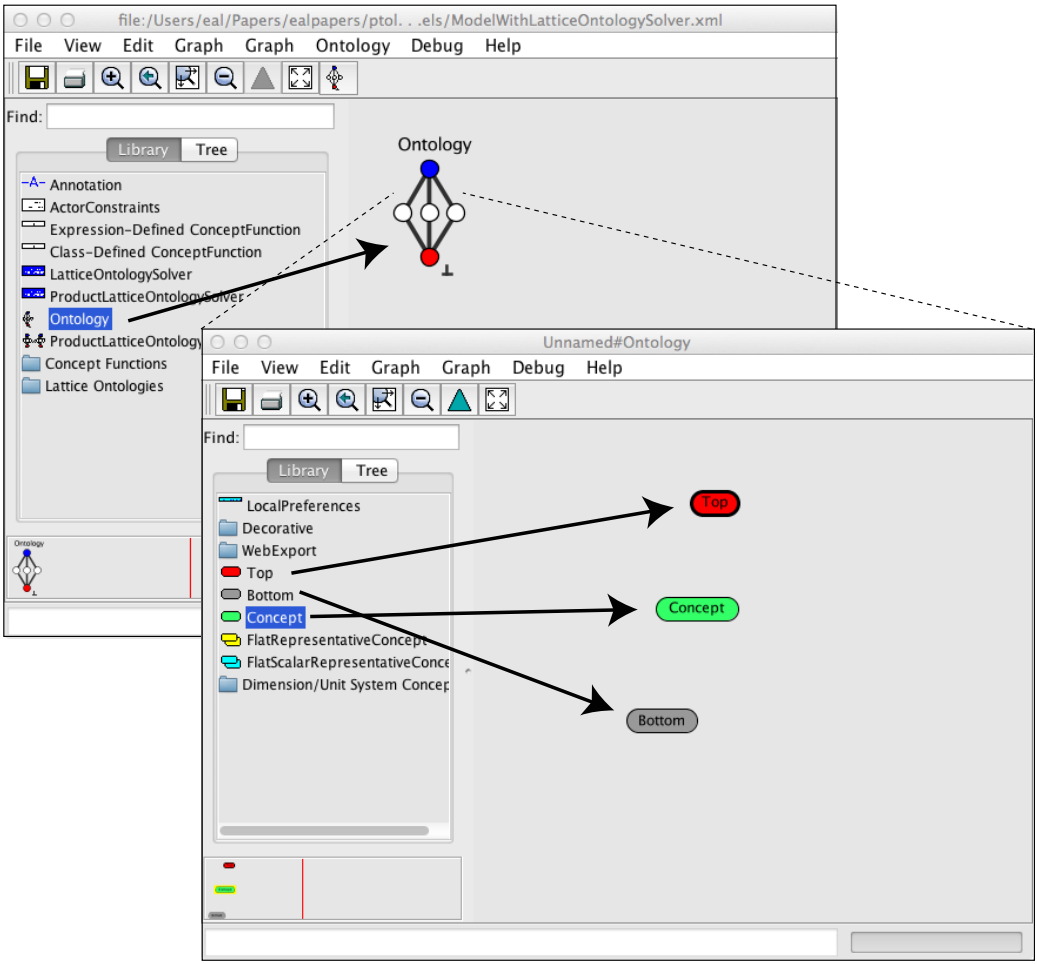


Figure 15.4: Steps in the construction of an ontology.

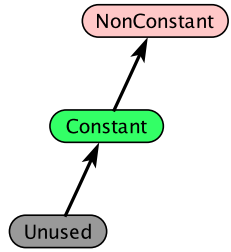


Figure 15.5: The lattice used for constant analysis.

cepts will be associated with ports in our model, and when *isAcceptable* is unchecked (the outline on the concept is bold), then it is an error in the model for any port to be associated with the concept. In our case, it is not an error for a port to be NonConstant, so we make the value of this parameter true.

Here, we include not only the concepts of Const and NonConst, but also explicitly include a notion of Unused. This concept will be associated with ports that are simply not participating in the analysis. We will use the NonConstant concept to represent any signal that may or may not be constant, so a better name might be PotentiallyNonConst or NotNecessarilyConst.

The last step in building an ontology is to establish relationships between the concepts. Do this by holding the control key (command key on a Mac) and dragging from the lower concept to the higher one. In this case, the relation between Constant and NonConstant is a **generalization relation**. The relations define an order relation between concepts (see sidebar on page 513), where if the arrow goes from concept a to concept b, then $a \leq b$. In this case, Constant $\leq$ NonConstant, and Unused $\leq$ Constant.

The meanings of these relations can vary with ontologies, but it is common for them to represent subclassing, a subset relation, or as an "is a" relation. In the subset interpretation, a concept represents a set, and concept A is less than another B if everything in A is also in B. The "is a" relation interpretation is similar, but doesn't require a formal notion of a set. E.g., if the concept A represents "dog" and the concept B represents "mammal," then it is reasonable to establish on ontology where $A \leq B$ under the "is a" interpretation. A dog is a mammal.

Example 15.3: In Figure 15.5, NonConstant represents anything that may or may not be constant. Hence, something that is actually constant "is a" NonConstant.

The interpretation of the bottom element, Unused in the example, is a bit trickier. Presumably, anything that makes no assertion about whether it is constant or not "is a" NonConstant. Indeed, since the order relation is transitive, this statement is implied by Figure 15.5. But why make a distinction between Unused and NonConstant? We could build an ontology that makes no such distinction, but in such an ontology, the inference engine would infer Constant for any port that imposes no constraints at all. This is probably an error. The bottom element, therefore, is used to indicate that the inference engine has no usable information at all. If a port resolves to Unused, then it is not playing the game. If we wish to force all ports to play the game, then we should set the *isAcceptable* parameter of Unused to false.

The Ptolemy II type system is an ontology, as shown in Figure 14.4. Here, the order relations represent subtyping, which in the case of Ptolemy II is based on the principle of lossless type conversion.

With the concepts as nodes and the relations as edges, the ontology forms a mathematical graph. The structure of this graph is required to conform with that a mathematical lattice (see sidebar on page 513). Specifically, the structure is a lattice if given any two concepts in the ontology, these two concepts have a least upper bound and a greatest lower bound. In this case, conformance is trivial. For example, the least upper bound of Constant and NonConstant is NonConstant. The greatest lower bound is Constant. But it is easy to construct an ontology that is not a lattice.

Example 15.4: Figure 15.6 shows an ontology that is not a lattice. Consider the two concepts, Dog and Cat. There are three concepts that are upper bounds for these two concepts, namely Pet, Mammal, and Animal. But of those three, there is no least upper bound. A least upper bound must be less than all other upper bounds.

If you build an ontology that is not a lattice, then upon invoking the solver, you will get an error message.

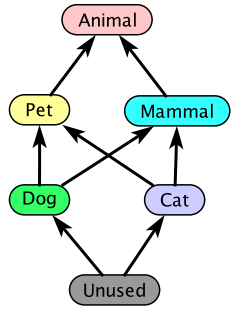


Figure 15.6: An ontology that is not a lattice.

15.1.2 Creating Constraints

Now that we have an ontology, to use it, we need to create constraints on the association between objects in the model and the concepts in the ontology. The most straightforward way to do this is with manual annotations in the model. With large models, however, this technique does not scale well. A more scalable technique is to define constraints that apply broadly to all actors of a class, and to specify default constraints that apply when no other constraints are specified. We begin with the manual annotations, because they are conceptually simplest.

Manual Annotations

The simplest way to relate objects in a model to concepts in the ontology is through manual annotations. Manual annotations take the form of inequality constraints. To create such constraints, find the **Constraint** annotation in the `MoreLibraries→Ontologies` library, and drag it into the model. Then specify an inequality constraint of the form `object >= concept` or `concept >= object`, where `object` is an object in the model (a port or parameter) and `concept` is a concept in the ontology.

Example 15.5: Figure 15.7 elaborates the model of Figure 15.3 by adding four annotations. Each of these has the form

```
port >= concept.
```

Specifically, the output ports of each Const actor are constrained to be greater than or equal to Constant, whereas the output port of the Ramp actor is constrained to be greater than or equal to NonConstant. The latter constraint, in effect, forces the port to NonConstant, since there is nothing greater than NonConstant in the ontology. The former constraints could be elaborated to force the ports to resolve to Constant by adding the complementary inequality, like

```
Constant >= Const.output.
```

However, this additional constraint is unnecessary. The solver will find the *least* solution that satisfies all the constraints, so in this case, since there are no other constraints on the output ports of the Const actors, they will resolve to Constant anyway.

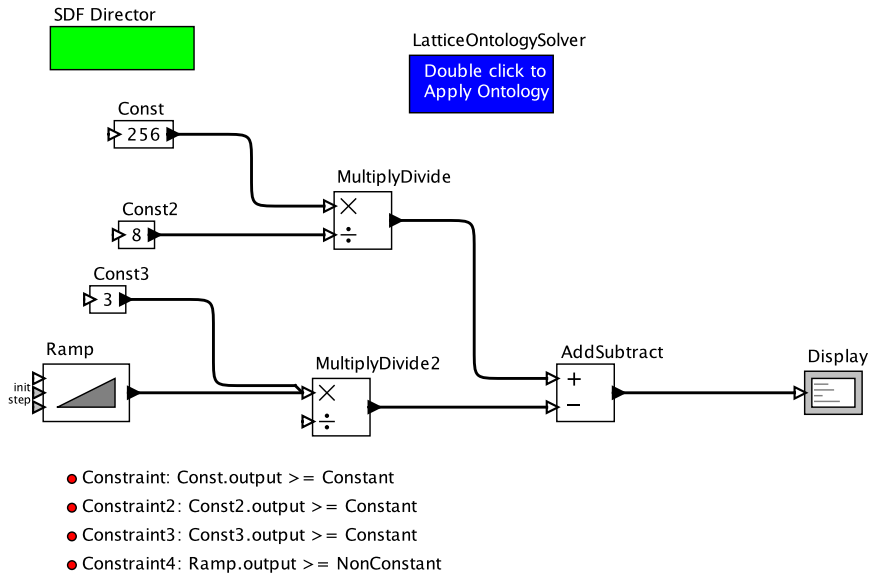


Figure 15.7: A model with manual constraints, using the ontology in Figure 15.5.

Once we have created all the constraints needed in our model, the next step is to run the analysis. The analysis can be run by right-clicking on the LatticeOntologySolver and selecting `Resolve Concepts`, as shown in Figure 15.8. Because this is a common operation, double-clicking on the LatticeOntologySolver will also run the analysis.

Example 15.6: Figure 15.8 includes the results of running the analysis. Notice that each port is annotated with the concept that it is now associated with. In addition, the port is highlighted with the same color specified for the concept in the ontology of Figure 15.5. The output ports of the Const actors, as expected, have resolved to Constant, and the the output of the Ramp has resolved to NonConstant. But more interestingly, downstream ports have also resolved in a reasonable way. The output of the MultiplyDivide is Constant (because both its inputs are Constant), and the the output of MultiplyDivide2 is NonConstant (because one of its inputs is NonConstant). These results are due to default constraints associated with actors, which as explained below, can be customized in an ontology.

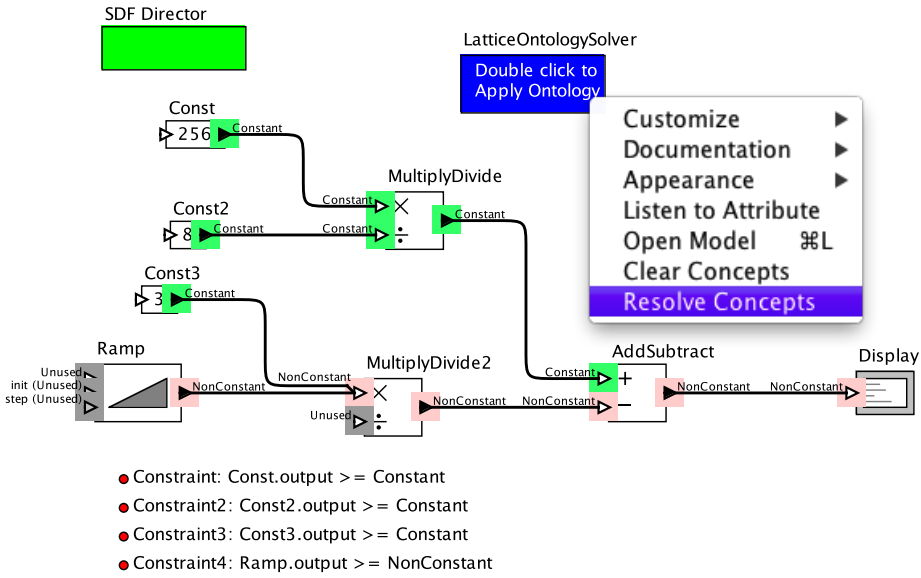


Figure 15.8: Running an analysis. [online]

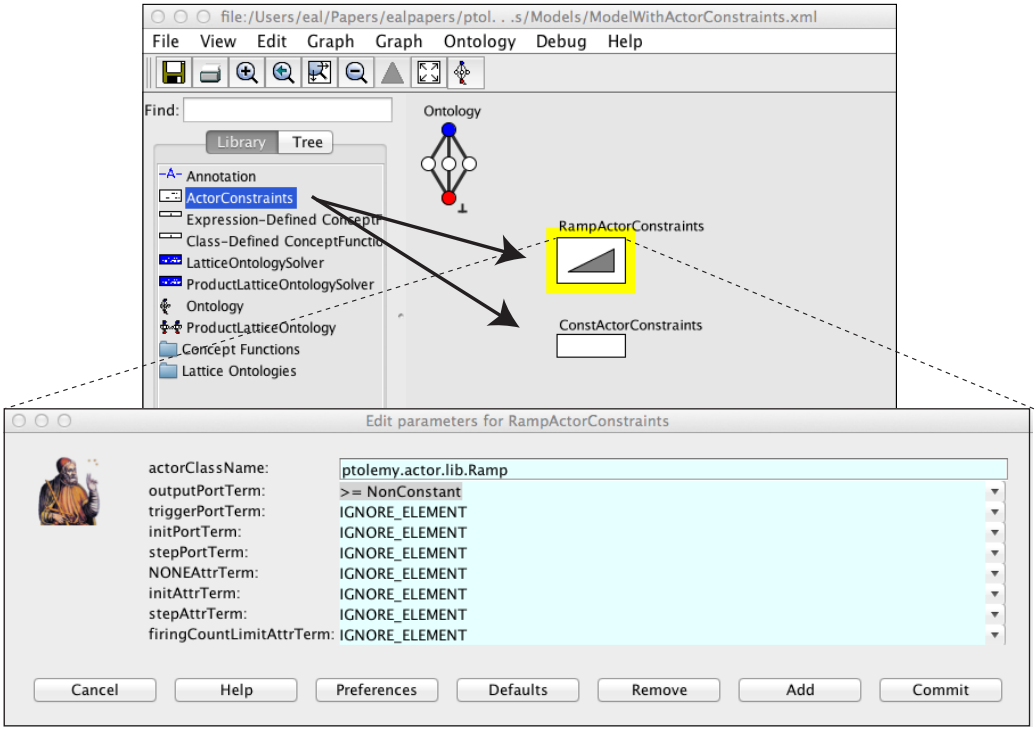


Figure 15.9: Adding actor constraints to an ontology.

Actor-level Constraints

The manual annotations in Figure 15.7 could become very tedious to enter in a large model. Fortunately, there is a convenient shortcut. As part of defining an ontology, we can specify constraints that will be associated with all instances of a particular actor class. Figure 15.9 shows how to do this. Inside the LatticeOntologySolver, we add one instance of **ActorConstraints** for each actor for which we want to specify default constraints.

Setting the *actorClassName* parameter of the ActorConstraints to the fully qualified class name of the actors to be constrained causes both the name and the icon of the instance of ActorConstraints to change to match the class of actors that it will constrain.

Example 15.7: Figure 15.9 shows two instances of ActorConstraints that have been dragged into the LatticeConstraintsSolver. The top one has the *actorClassName* parameter set to `ptolemy.actor.lib.Ramp`, the class name of the Ramp actor. (To see the class name of an actor, linger over it in Vergil.)

Once the class name has been set, upon re-opening the parameter dialog, a new set of parameters will have appeared, one for each port belonging to instances of the actor class, and one for each parameter of the actor.

Example 15.8: Figure 15.9 shows these parameters for the Ramp actor. Here we can see that constraints have been set that will force the *output* port to be greater than or equal to NonConstant, and that all other constraints have been set to `IGNORE_ELEMENT`. Once a similar constraint has been added to the ConstActorConstraints component, constraining the output ports of instances of Const to be `>= Constant`, then the four constraints at the bottom of Figure 15.7 are no longer necessary. They could be removed, and the results of the analysis will be the same as in Figure 15.8.

The constraints associated with a port or parameter can take any one of the following forms:

- `NO_CONSTRAINTS` (the default)
- `IGNORE_ELEMENT`
- >= *concept*
- <= *concept*
- == *concept*

By default, the ActorConstraints actor will set `NO_CONSTRAINTS` as the constraint of each port and parameter. This means that instances of the actor allow their ports and parameters to be associated with any concept. In this ontology, the association will always result in Unused, so we could equally well have left all the constraints at the default `NO_CONSTRAINTS`, except those for the output ports.

Default Constraints

Notice in Figure 15.8 that not only have the outputs of the Const and Ramp actors resolved to the appropriate concepts, but so have those of downstream actors, including MultiplyDivide and AddSubtract. How did this come about?

With respect to this analysis, both the MultiplyDivide actor and the AddSubtract actors behave the same way. Given only constant inputs, they produce a constant output, but given any non-constant input, they produce a (potentially) non-constant output. In terms of the ontology lattice (Figure 15.5), the output concept will always be greater than or equal to all of the input concepts. In other words, the output should be constrained to be greater than or equal to the least upper bound of all the inputs. It turns out that this type of inference behavior is a very common one. For this reason it is the default constraint for all actors. Actors that do not have explicit constraints will inherit this default constraint. This means that we can omit specifying any more ActorConstraints, since the global default constraint is sufficient for all of our remaining actors.

15.1.3 Abstract Interpretation

Identifying signals as either constant or non-constant is a particularly simple form of **abstract interpretation** (Cousot and Cousot, 1977). In abstract interpretation, instead of actually computing the values of variables, we classify the variables in more abstract terms, such as whether their values vary over time. Ontologies can be used to systematically apply more sophisticated abstractions, determining for example whether variables are always positive, negative, or zero. This can be used to expose errors in designs, and also to optimize design by removing unnecessary computations.

> **Example 15.9:** The model in Figure 15.10 produces a constant stream of zeros. Were we to apply the same Constant-NonConstant analysis as before to this model, we would be able to determine that the output is constant. But it would not tell us that the output is a constant stream of *zeros*.

To address this problem, we can use the more elaborate ontology shown in Figure 15.11. This ontology is used to abstract numeric variables as positive, negative, or zero-valued

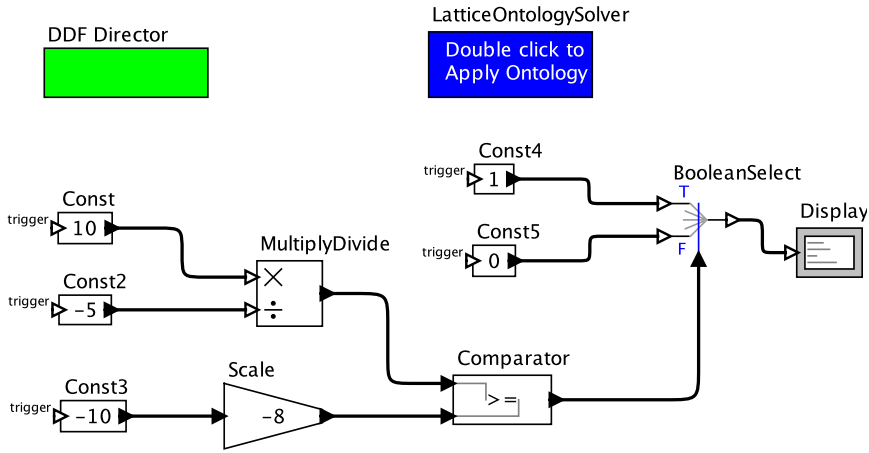


Figure 15.10: A model that produces only zeros.

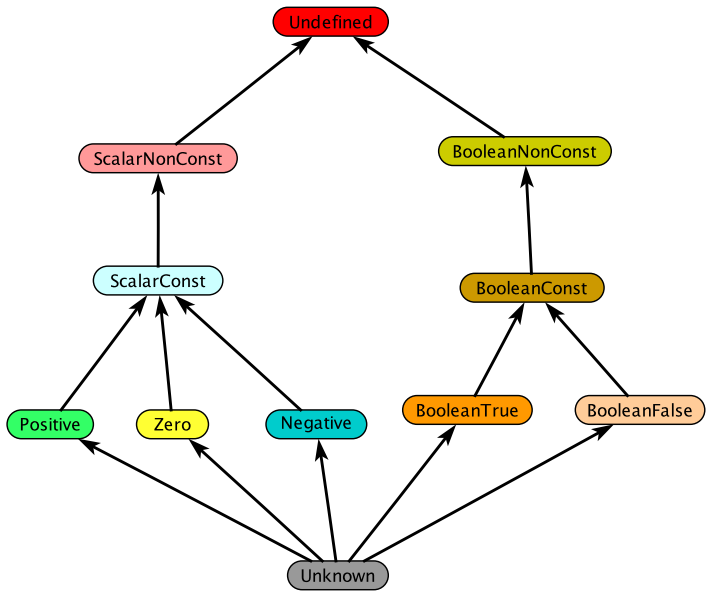


Figure 15.11: An ontology that tracks the sign of numeric variables and the value of boolean variables.

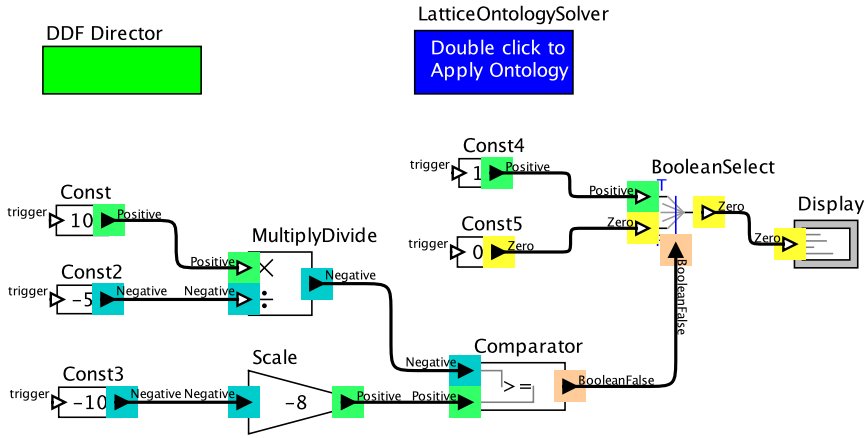


Figure 15.12: Result of applying an analysis based on the ontology in Figure 15.11 to the model in Figure 15.10. [online]

numbers, in addition to whether they are constant or non constant. For boolean-valued variables, if the variable is constant, then it also tracks whether the constant is true or false. With appropriate actor constraints, this ontology can be used to produce the result shown in Figure 15.12, which determines that the output is a constant stream of zeros.

15.2 Finding and Minimizing Errors

In this section, we discuss how to use an analysis to identify errors, and discuss tools available to help in correcting the errors. For this discussion, let us consider the ontology in Figure 15.13. This ontology models physical dimensions. Specifically, it distinguishes the concepts of time, position, velocity, and acceleration. We will show that with appropriate actor constraints, it can use properties of these dimensions in inference.

Example 15.10: Figure 15.14 shows a piece of a larger model that shows interesting inference of dimensions. This is a model of a car with a **cruise control**, where the input is a desired speed, and the outputs are acceleration, speed, and position. In this model, when velocity is divided by time, the result is acceleration. When

acceleration is integrated over time, the result is velocity. When velocity is integrated over time, the result is position. This analysis relies on actor constraints for arithmetic operations and integrators.

The ontology in Figure 15.13 explicitly includes the concepts Unknown and Conflict. The difference between Unknown and Conflict is subtle and deserves mention. Conflict

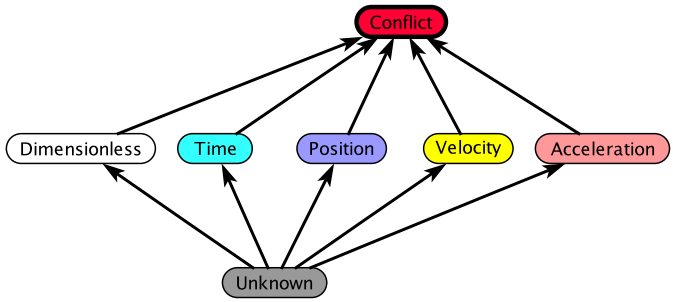


Figure 15.13: An ontology for analyzing physical dimensions.

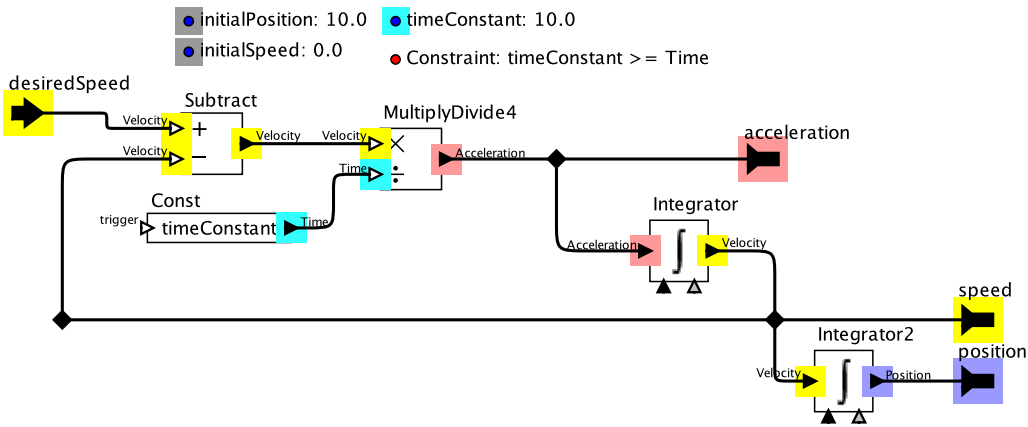


Figure 15.14: A piece of a larger model that shows interesting inference of dimensions. [online]

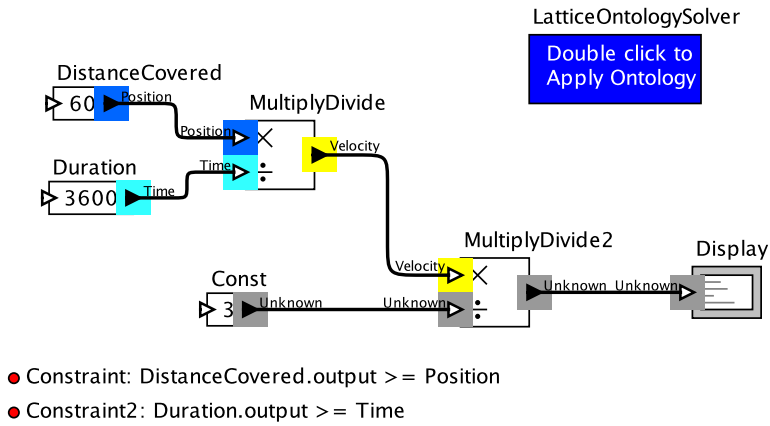


Figure 15.15: A model analyzing physical dimensions with too few annotations. Running the analysis reveals where additional annotations are needed. [online]

represents a situation where the analysis has detected that a given signal cannot have any of the dimensions; thus, the analysis has found an error in the model due to conflicting use of dimensions. For this reason, in this ontology, the *isAcceptable* parameter of Conflict is set to false, resulting in a bold outline in Figure 15.13. In addition, if any port resolves to Conflict, then running the analysis will report an error.

In the ontology in Figure 15.13, Unknown represents a case where the analysis cannot say conclusively anything about the given signal; this means that the property being analyzed cannot be proved with the given assumptions. Unknown in this case plays a similar role as Unused in Figure 15.5. When a port resolves to Unknown, this can point to there not being enough constraints in the model. This may or may not be an error, so *isAcceptable* is left at its default value of true.

Example 15.11: An example of an underconstrained model is shown in Figure 15.15. Here, the model divides a velocity by a time to get an acceleration, but the constraint specifying the dimension of the time value produced by the Const actor has been omitted. When running the analysis, we get results shown the figure. The lack of information propagates throughout the model. This can be fixed, of course, by adding a manual annotation to the model as shown in Figure 15.16.

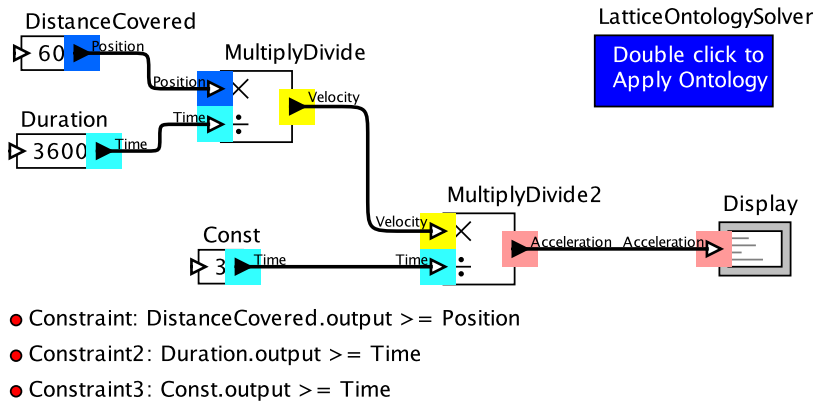


Figure 15.16: Adding an additional constraint allows for a complete analysis. [online]

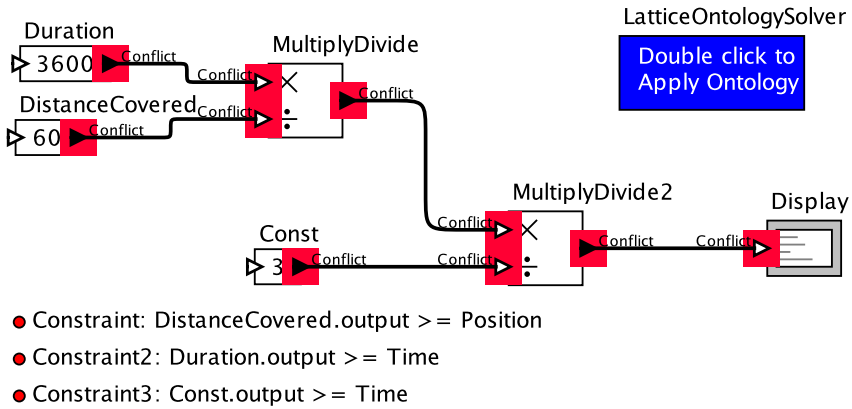


Figure 15.17: An example model with conflicting dimensions due to an error in the model. Running the analysis on this model shows that the whole model is in conflict. [online]

In general, overconstrained models can be more difficult to deal with.

Example 15.12: An example of an overconstrained model is shown in Figure 15.17. Here, the model builder has incorrectly divided a time by a position, where presumably the reverse operation was intended. The result is conflicts throughout the model.

The conflict in the previous example arises because of the ActorConstraints that are defined for the MultiplyDivide actor as part of the ontology (see Figure 15.9). Specifically, the ontology gives for the *outputPortTerm* the following expression:

```
>=
(multiply == Unknown || divide == Unknown) ? Unknown :
(multiply == Position && divide == Time) ? Velocity :
(multiply == Velocity && divide == Time) ? Acceleration :
(multiply == Position && divide == Velocity) ? Time :
(multiply == Velocity && divide == Acceleration) ? Time :
(divide == Dimensionless) ? multiply :
Conflict
```

This constrains the output concept as a function of the input concepts. If the *multiply* input is Position and the *divide* input is Time, for example, this expression evaluates to Velocity. If *divide* input is Dimensionless, then it evaluates to whatever *multiply* is. If *multiply* is Time and *divide* is Position, it evaluates to Conflict.

In the example of Figure 15.17, notice that conflicts propagate upstream as well as downstream. In this example, we have set the *solverStrategy* of the LatticeOntologySolver to `bidirectional`.* The default value of this parameter is `forward`, which means that when there is a connection from an output port to an input port, then the concept associated with the input port is constrained to be greater than or equal to the concept associated with the output port. When the parameter value is set to `bidirectional`, then the two concepts are required to be equal. With the `bidirectional` setting, constraints propagate upstream in a model just as easily as downstream. So although this is reasonable to

*Since double clicking on the LatticeOntologySolver runs the analysis, double clicking cannot be used to access the parameters. Instead, hold the alt key while you click to access the parameters.

do for dimension analysis, it makes it difficult to see which part of the model is causing the error.

By default, the solver finds the least solution that satisfies all the constraints.[†] Hence, the way that our analysis deals with conflicting information is to promote signals to the least upper bound of the conflicting concepts (or greatest lower bound when computing greatest fixed points).

The analysis is able to correctly detect the error, as shown in Figure 15.17, but there is a problem. Unlike the underconstrained case in which the error was relatively contained, in this case Conflict propagates throughout the model, making it very difficult to see where the source of the error is. In this simple example it is not too difficult to find the error, but as models grow, so does the difficulty in tracking down the source of an error of this type.

In order to address this problem, we have introduced an error minimization algorithm. This algorithm is implemented in the **DeltaConstraintSolver** actor, which is a subclass of LatticeOntologySolver. It can be found in the same `MoreLibraries→Ontologies` library. The DeltaConstraintSolver offers a `Resolve Conflicts` item in the context menu in addition to the `Resolve Concepts` that we used before.

Given a model in which at least one signal resolves to an unacceptable solution, the algorithm realized by `Resolve Conflicts` finds a subset of those constraints with the property that removing any additional constraint does not produce any error. Highlighting the result of running the analysis with these constraints will highlight only a subsection of the model that contains an error. In practice, this means in contrast to the full analysis, where the highlighted result shows many errors throughout the model as shown in Figure 15.17, the modified algorithm highlights only a single path through the model that contains an error, as shown in Figure 15.18. In this example, only the Duration, DistanceCovered, and MultiplyDivide actors (each of which is an instance of Const) are highlighted, since they are sufficient to cause the error. The unhighlighted actors (in this case the Const, MultiplyDivide2, and Display) are not necessary to cause the error, so the model builder can ignore them in trying to find the cause of the error.

In this case, the port that was highlighted with the erroneous concept (Conflict) was the location of the error, but in general, this need not be the case. The only guarantee that is made is that there is an error somewhere in the path of all highlighted signals. This

[†]This can be changed by changing the *solvingFixedPoint* parameter of the LatticeOntologySolver from the default `least` to `greatest`. In that case, the solver will find the highest solution in the lattice that satisfies all the constraints.

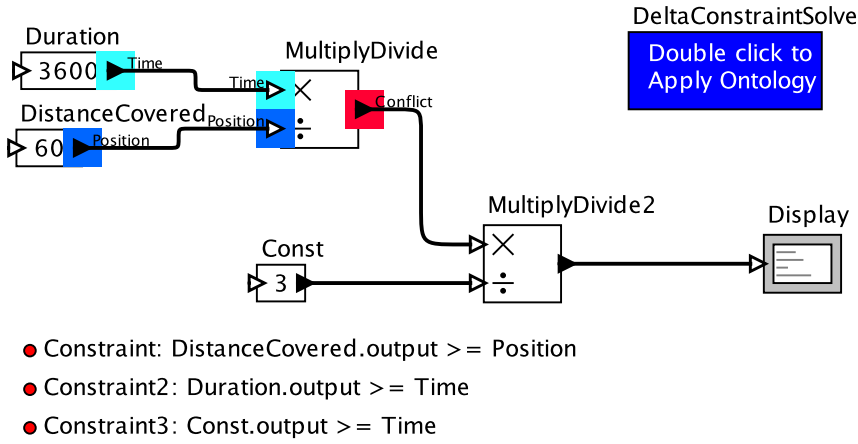


Figure 15.18: With our error minimization algorithm, finding errors is easier. [online]

means that in general, all of the signals that are highlighted may need to be checked in order to find the source of the error. The gain of this technique comes from the many unhighlighted actors in the model that can be completely ignored.

15.3 Creating a Unit System

In the previous section, we saw an analysis that checked that the dimensions of a system were being used in a consistent way, to avoid errors such as integrating a velocity and expecting to get an acceleration value out. A similar but more insidious type of error is when two signals have the same dimensions, but different units, such as feet vs. meters, or pounds vs. kilograms. Since this is a more subtle problem – results can deviate by only a small scaling factor – it is even more desirable to check these types of properties automatically. Section 13.7 describes a built-in unit system provided in Ptolemy II. But a units system is just another ontology, and the ontology framework can be used to create specialized units systems.

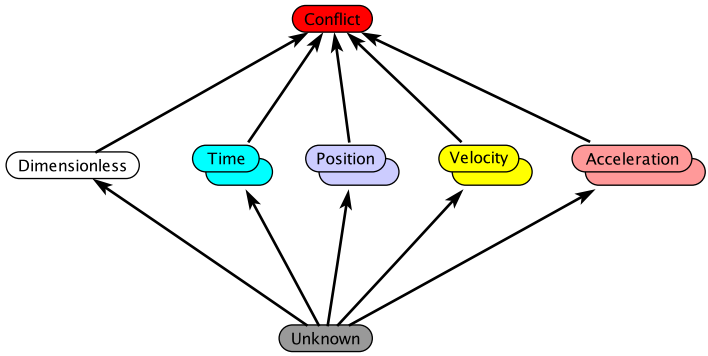


Figure 15.19: A unit system for physical units.

15.3.1 What are Units?

In order to discuss what units are, we first need to discuss how units differ from one another. There are two ways that units can differ. They can be different measures of the same quantity, like feet and meters, or they can be measures of fundamentally different quantities, like feet and seconds. Differences of the second type are exactly what is captured in the dimension ontology, so our approach to units will be similar, but extended to deal with different units within a single dimension.

Ptolemy II supports creating a **units system ontology** with the same freedom as other ontologies. Users can choose the units and dimensions that are appropriate to analyze the model at hand, and thus make the ontology only as complicated as it needs to be to perform the desired analysis. Since we have already discussed a physical dimension ontology that had concepts for Position, Velocity, Acceleration, etc., we will continue to use an example unit ontology where the units are drawn from the same dimensions, as shown in Figure 15.19. However, instead of building our ontology using only simple instances of Concept, we use instances of **DerivedDimension** and **Dimensionless**. These are concepts specialized to support units, and are selected from the `Dimension/Unit System Concepts` sublibrary provided by the ontology editor (see Figure 15.4).

As shown in Figure 15.19, DerivedDimension has a slightly different icon, a double oval, suggestive that it is in fact a **representative concept**, a single object that stands in for a family of concepts. DerivedDimension has some parameters that we set and parameters

that we must add to build a unit system. We outline what this looks like here, but to build a unit system from scratch, you will need to refer to the documentation.

15.3.2 Base and Derived Dimensions

The dimensions from which units are drawn are split into two types, **base dimensions**, and **derived dimensions**. Base dimensions are dimensions that cannot be broken down into any smaller components, such as Time, whereas derived dimensions can be expressed in terms of other dimensions. For example, Acceleration can be expressed in terms of Position and Time. Note that there is no restriction on which dimensions can be base dimensions or derived dimensions. There is no technical reason that a model builder could not define Acceleration to be a base dimension and derive Position, although it is not the natural choice in this situation. When defining a base dimension, the user only needs to specify the units within that dimension. This is done by adding parameters to the DerivedDimension concept.

Example 15.13: Figure 15.20 shows parameters that have been added to the Time dimension of Figure 15.19. Here, the user specifies the scaling factor of all units

secFactor:	1.0
hrFactor:	3600*secFactor
dayFactor:	24*hrFactor
sec:	{ Factor = secFactor }
ms:	{ Factor = 0.001*secFactor }
us:	{ Factor = 1E-06*secFactor }
ns:	{ Factor = 1E-09*secFactor }
minute:	{ Factor = 60*secFactor }
hr:	{ Factor = hrFactor }
day:	{ Factor = dayFactor }
yrCalendar:	{ Factor = 365.2425*dayFactor }
yrSidereal:	{ Factor = 31558150*secFactor }
yrTropical:	{ Factor = 31556930*secFactor }

Figure 15.20: An example of the specification of the Time base dimension.

within that dimension with respect to one another, using named constants as needed to make the calculations clearer.

The Time dimension is a base dimension. The definition of a derived dimension is slightly more involved, since a derived dimension must explicitly state which dimensions it is derived from. Derived dimensions must specify both how the derived dimension is derived from other dimensions, and how each of its individual units are derived from units of those other dimensions.

Example 15.14: Figure 15.21 shows an example of the specification of the Acceleration dimension of Figure 15.19. Here, the first lines show that an Acceleration is built from the Time and Position base dimensions, and that an acceleration has units of $position/time^2$. The rest of the specification shows how individual units of acceleration are related to units of position and time.

The main benefit of specifying units this way is that we can infer the constraints for multiplication, division, and integration, which are used in many actors.

```
dimensionArray:  { {Dimension = "LengthConcept", Exponent = 1}, {Dimension = "TimeConcept", Exponent = -2} }
LengthConcept:   Position
TimeConcept:     Time
m_per_sec2:      { LengthConcept = {"m"}, TimeConcept = {"sec", "sec"} }
cm_per_sec2:     { LengthConcept = {"cm"}, TimeConcept = {"sec", "sec"} }
ft_per_sec2:     { LengthConcept = {"ft"}, TimeConcept = {"sec", "sec"} }
kph_per_sec:     { LengthConcept = {"km"}, TimeConcept = {"hr", "sec"} }
mph_per_sec:     { LengthConcept = {"mi"}, TimeConcept = {"hr", "sec"} }
```

Figure 15.21: An example of the specification of the Acceleration derived dimension.

15.3.3 Converting Between Dimensions

In the case that units are used inconsistently, there is a error in the model. Not all units errors are the same, however. Units errors that result from interchanging signals with different dimensions point to a model whose connections must be changed, but unit errors from interchanging signals with different units of the same dimensions can be fixed by simply converting the units from one form to another. While it is technically possible to perform this type of conversion automatically, Ptolemy does not do this. Unit errors are an error in modeling, and model errors should never be concealed from model builders.

In keeping with this philosophy, a **UnitsConverter** actor[‡] must be explicitly added to a model in order to perform conversion between two units of the same dimension. This conversion happens both during the analysis, when the actor creates constraints on the units of its input and output ports, and during runtime, when the actor performs the linear transform from one unit to the other. Since the ontology knows the conversion factors between components, model builders need only specify the units to convert between, rather than the logic for conversion that would need to be specified in order to do conversion completely manually.

The parameters of the UnitsConverter actor, shown in Figure 15.22, include the *unitSystemOntologySolver*, which refers to the name of the LatticeOntologySolver of the units system analysis. (The name in this case is DimensionAnalysis.) Since it is only possible

[‡]which can be found in `MoreLibraries→Ontologies`

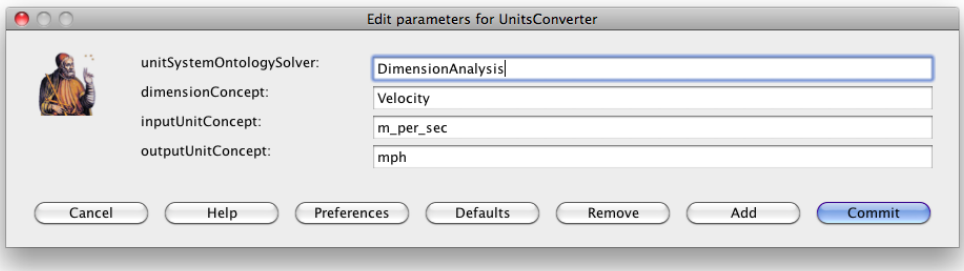


Figure 15.22: Using the UnitsConverter requires setting the name of the associated solver, as well as the input and output units.

to convert between units of the same dimension, the dimension is only specified once, in the *dimensionConcept* parameter, and the individual units of the input and output are specified in the *inputUnitConcept* and *outputUnitConcept* parameters respectively.

Example 15.15: Assume that we have a model that makes the unit error shown in Figure 15.23, where a velocity is provided in meters per second rather than miles per hour. By adding in a UnitsConverter as shown in Figure 15.24, and setting the parameters to convert from Velocity_mph to Velocity_m_per_sec as shown in Figure 15.22, we can create a model that both passes our units analysis and performs the conversion at runtime. Since the ontology analysis is not required to pass before running a model, the version of the model in Figure 15.23 without the UnitsConverter actor can still run. It produces an incorrect output value, however, since it treats the velocity value in Const as expressed in miles per hour as one in meters per second.

15.4 Summary

One of the key challenges in building large, heterogeneous models is ensuring correct composition of components. The components are often designed by different people, and their assumptions are not always obvious. Customized, domain-specific ontologies offer a powerful way to make assumptions clearer. Applying such ontologies in practice, however, can be very tedious, because they typically require the model builders to extensively annotate the model, decorating every element of the model with ontology information. The infrastructure described in this chapter leverages a very efficient inference algorithm that can significantly reduce the effort required to apply an ontology to a model. Far fewer annotations are required than is typical because most ontology associations are inferred.

Domain-specific ontologies and the associated constraints, however, can get quite sophisticated. The vision here is that libraries of ontologies, constraints, and analyses will be built up and re-used. This is certainly possible with the unit systems and dimension systems, but it also seems possible to construct libraries of analyses that are industry- or application-specific. Since these analyses are simply model components, they are easy to share among models within an enterprise.

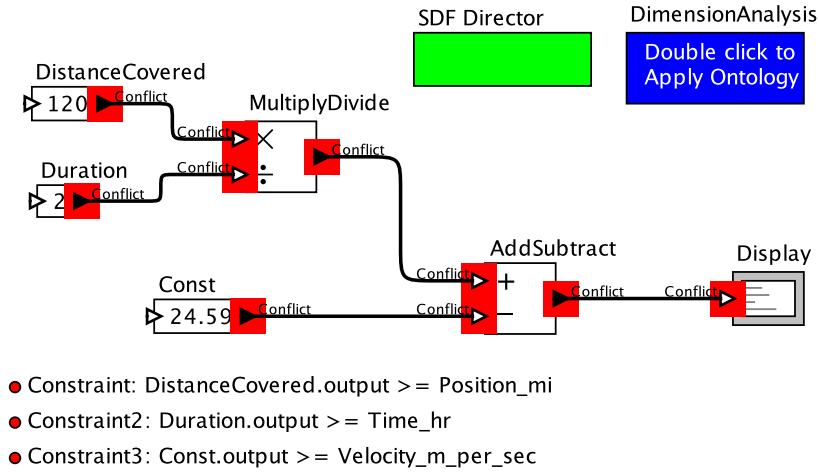


Figure 15.23: An example that adds a quantity in miles per hour to one in meters per second without conversion, resulting in conflicts throughout the model. [online]

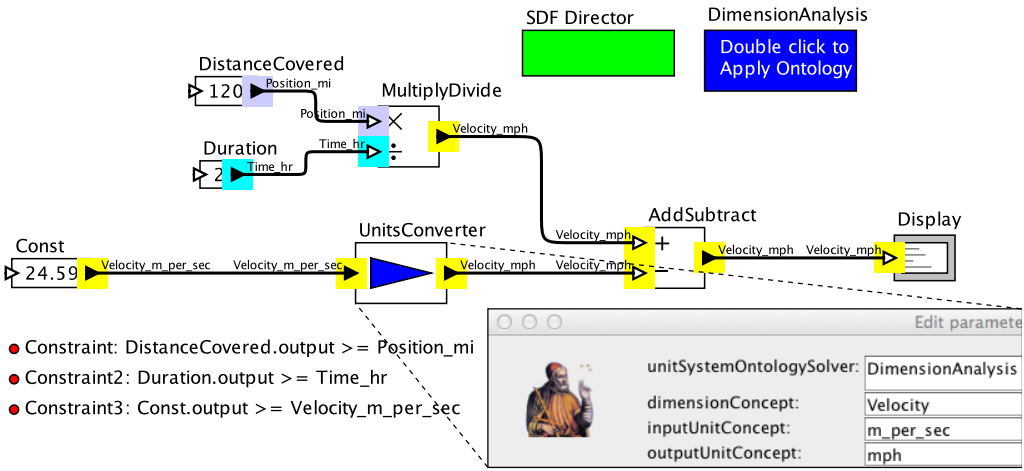


Figure 15.24: A model that uses the UnitsConverter actor to convert from meters per second to miles per hour. [online]

More interestingly, simple ontologies can be systematically combined to create more sophisticated ontologies, including ones where there are constraints that reference more than one ontology. For example, the ontology in Figure 15.11 could be factored into two simpler ontologies, once that refers to scalars and one that refers to booleans, and a product ontology could be defined in terms of these two simpler ontologies. The interested reader is referred to the Combining Ontologies chapter of Lickly (2012).

16 Web Interfaces

Christopher Brooks, Edward A. Lee, Elizabeth A. Latronico, Baobing Wang, and Roxana Gheorghui

Contents

Ptolemy II includes a flexible mechanism for creating web pages from models and for building web services. The more basic mechanism is the **export to web**, which simply makes a model available as a web page for browsing using a web browser. Such a web page provides easy access and documentation for models that archives both the structure of the models and the results of executing the models. It can be used to share information about models or their execution without requiring installation of any software, since an ordinary web browser is sufficient. More interestingly, the mechanism is extensible and customizable, allowing for creation of fairly sophisticated web pages. You can associate

hyperlinks or actions defined in JavaScript* with icons in a model. The customization can be done for individual icons in a model or for sets of icons in a model.

The more advanced mechanism described in this chapter turns a model into a web service. The machine on which the model executes becomes a web server, and the model defines how the server reacts to HTTP requests that come in over the Internet. A web service can be created that does anything that can be done in a Ptolemy II model. Some care is required, of course, to ensure that such a web service does not create unacceptable security vulnerabilities for the web server machine.

16.1 Export to Web

To export a model to the web, select [`File→Export→Export to Web`], as shown in Figure 16.1. This will open a dialog that enables you to select a directory (or create new directory). That directory will be populated with a file called index.html, some image files, and some subdirectories. One image file shows whatever portion of the model is visible when you perform the export. In addition, there will be an image file for each open plot window. Moreover, there will be one subdirectory for each composite actor that is open at the time of export.

The export dialog offers a number of options, as follows.

- *directoryToExportTo*: The directory into which to put the web files. If no directory is given, then a new directory is created in the same directory that stores the MoML file for the model. The new directory will have the same name as the model, with any special characters replaced so that the name is a legal file name.
- *backgroundColor*: The background color to use for the image model. By default, this is blank, which means that the image will use whatever background color the model has (typically gray). But white is a good option for web pages, as shown in Figure 16.1.
- *openCompositesBeforeExport*: If this is true, then composite actors in the model are opened before exporting. Each composite actor will also be exported into its own web page, and hyperlinks will be created in the top-level image to allow navigation to those web pages in the browser. If you want only some of the composite actors to be included in the export, then you can manually open the ones you want. Only open windows will be included in the export.

*By default, the export to web facility uses JavaScript to display the parameters of actors. JavaScript may be disabled in your web browser. To enable JavaScript. See http://support.microsoft.com/gp/howtoscript.

- *runBeforeExport*: If this is true, then the model is run before exporting. This has the side effect of opening plot windows, which will therefore be included in the export. If you want only some of the plot windows to be included in the export, then you can run the model and close the ones you don't want. Only open plot windows will be included in the export.
- *showInBrowser*: If this is true, then once the export is complete, the resulting web page will be displayed in your default browser.
- *copyJavaScriptFiles*: If this is true, then additional files will be included in the exported page so that the page does not depend on any files from the internet. The files include

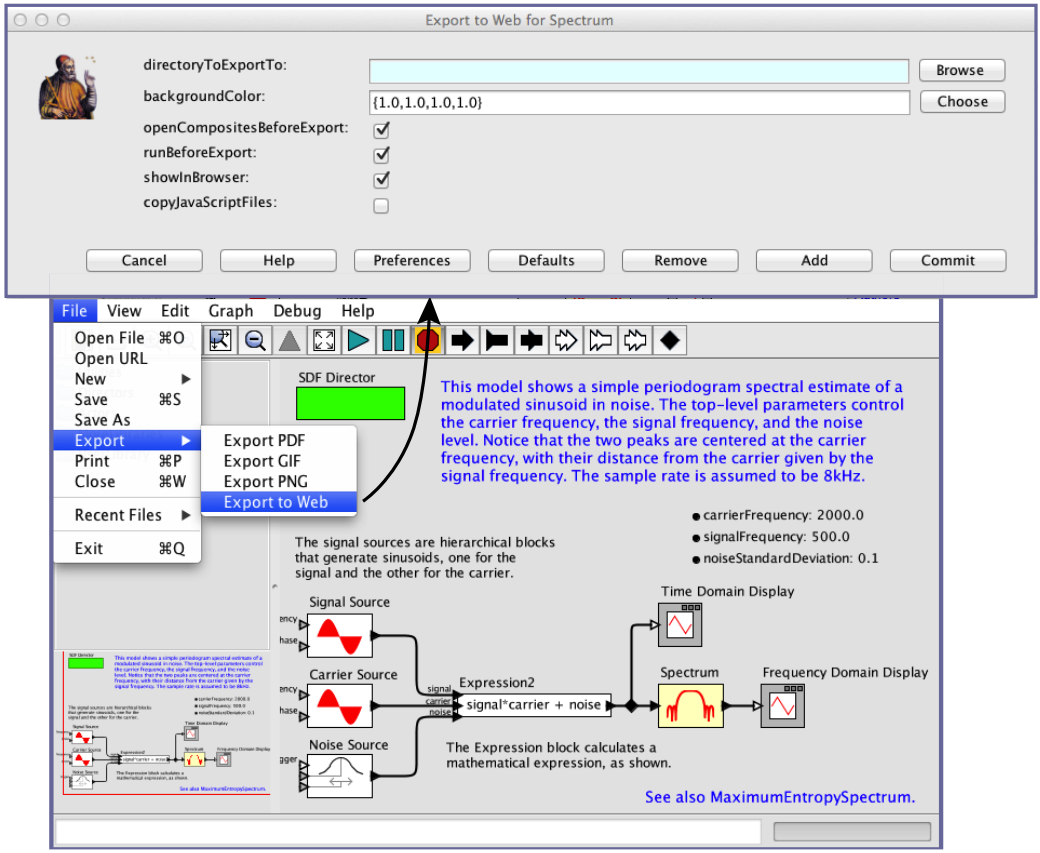


Figure 16.1: Menu command to export a model to the web.

JavaScript code and image files that affect the interactivity and look-and-feel of the web page. By default, these files are not included and are instead retrieved by the web page from `http://ptolemy.org`.

For the example shown in Figure 16.1, the resulting web page is displayed by the Safari web browser as shown in Figure 16.2. This page exhibits some of the default behavior of export to web. A title for the page is shown at the top; this is, by default, the name of the model. Moreover, in the image shown in Figure 16.2, the mouse is hovering over the Signal Source actor, which is outlined; when the mouse hovers over an actor, then by default, a table with the parameter values of the actor is displayed at the bottom of the page, as shown in Figure 16.2.

The generated web page shows the portion of the model visible in the viewing pane. Therefore, parts of the model can be hidden by resizing the viewing pane. For example, one might wish to hide a long list of parameters or attributes. Simply resize the pane, then perform the export.

In Figure 16.1, *openCompositesBeforeExport* and *runBeforeExport* are both set to true (the default is false). Hence, the model is executed before the export, opening plot windows. Hyperlinks to the plot windows are created, and clicking on a plot actor on the web page image will display the plot, as shown in Figure 16.3. In addition, the composite actors in the model, Signal Source, Carrier Source, and Spectrum, all have hyperlinks to a page showing the inner structure of the composite.

All these functions can be customized, as we will explain next.

16.1.1 Customizing the Export

As shown in Figure 16.4, the `Utilities→WebExport` library provides attributes that, when dragged into a model, customize the exported web page. This section explains each of the items in this library, shown on the left in the figure. In each case, you can right click (or control click on a Mac) and select `Get Documentation` to view documentation about the attribute. The attributes are related to one another as shown in the UML class diagram in Figure 16.5.

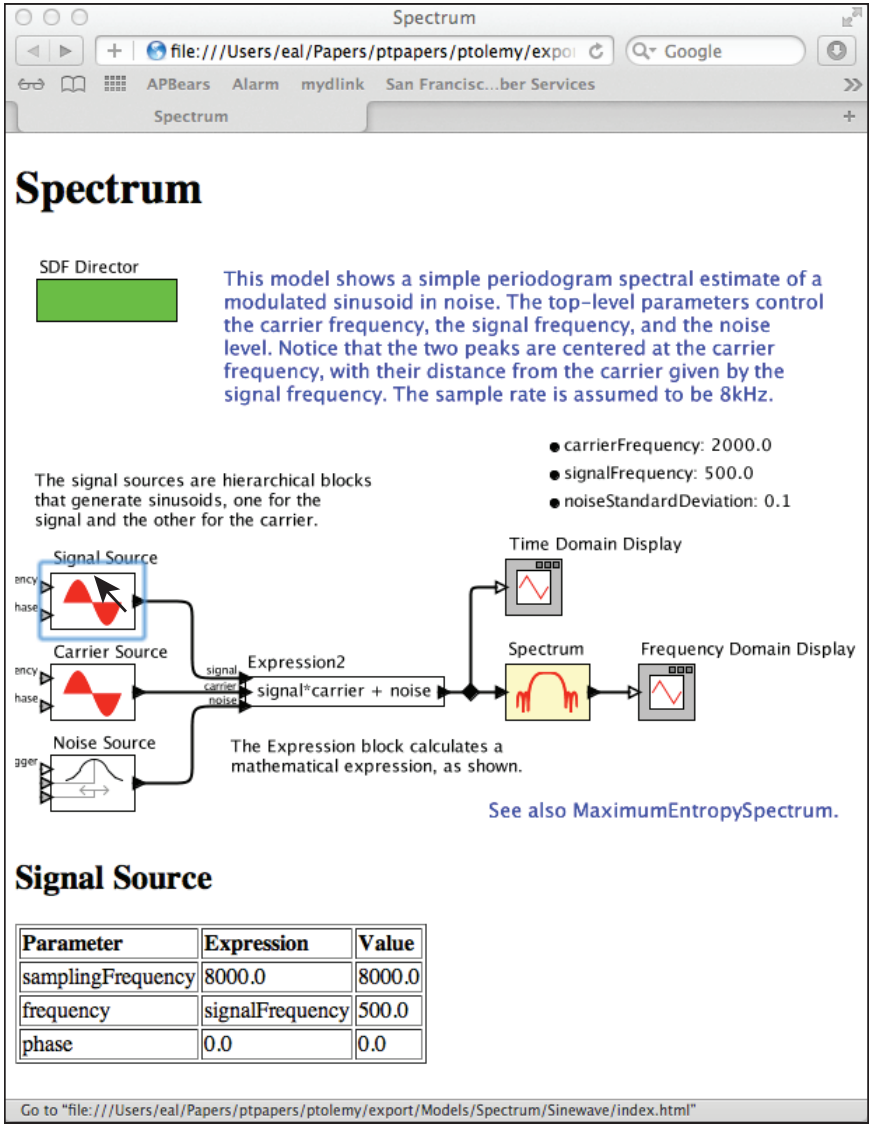


Figure 16.2: Web page exported from the model shown in Figure 16.1.

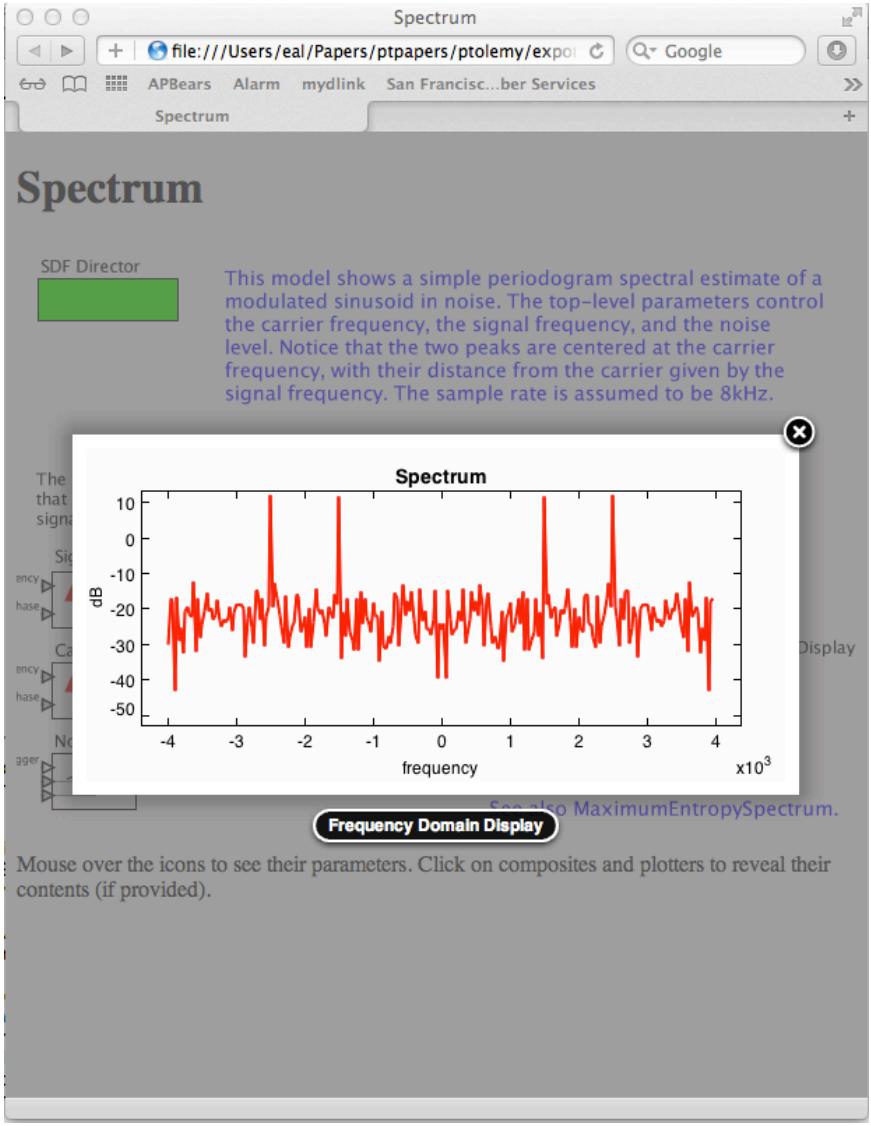


Figure 16.3: Clicking on the Frequency Domain Display actor in Figure 16.2 displays the plot generated by running the model.

HTMLText: Adding Text to Web Pages

The *HTMLText* attribute inserts HTML text into the page exported by Export to Web. Drag the attribute onto the background of a model, as shown in Figure 16.4, and double click on its icon to specify the HTML text to export. To specify the text to include in the HTML page, double click on the icon for the *HTMLText* attribute (which by default is a textual icon reading "Content for Export to Web"), as shown in Figure 16.6. You can type in the text to export, including any HTML content you like such as hyperlinks and

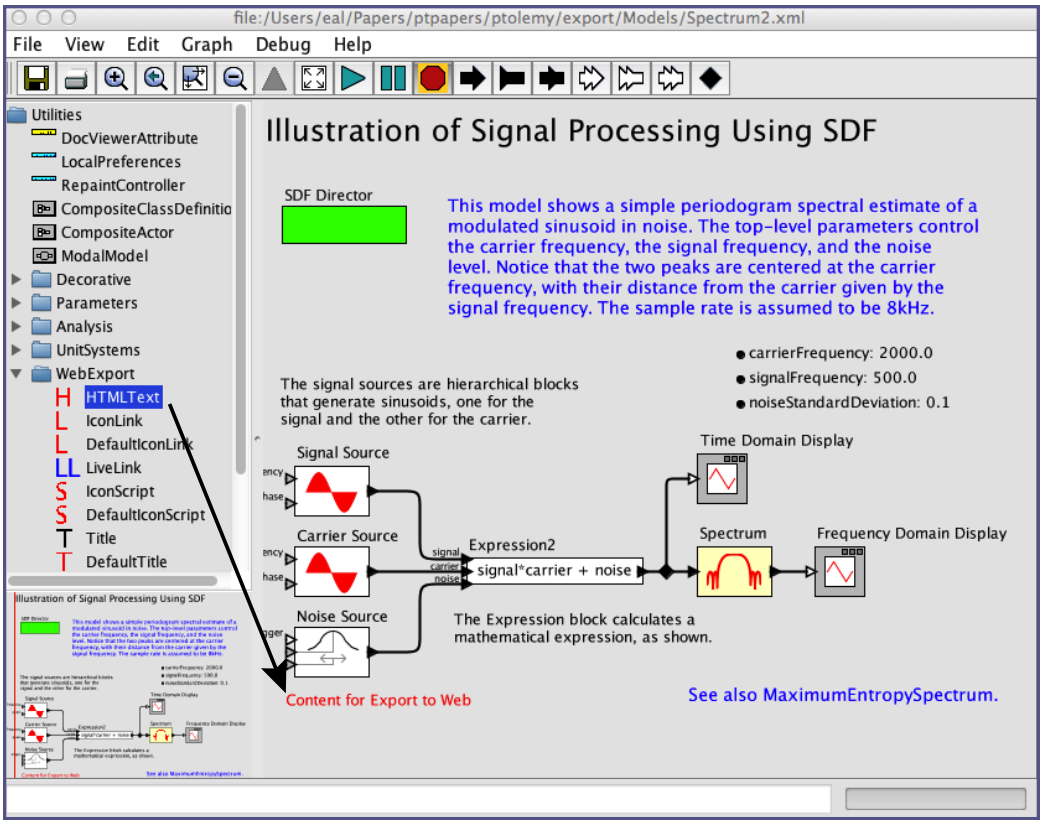


Figure 16.4: The `Utilities→Web Export` library provides attributes that, when dragged into a model, customize the exported web page. [online]

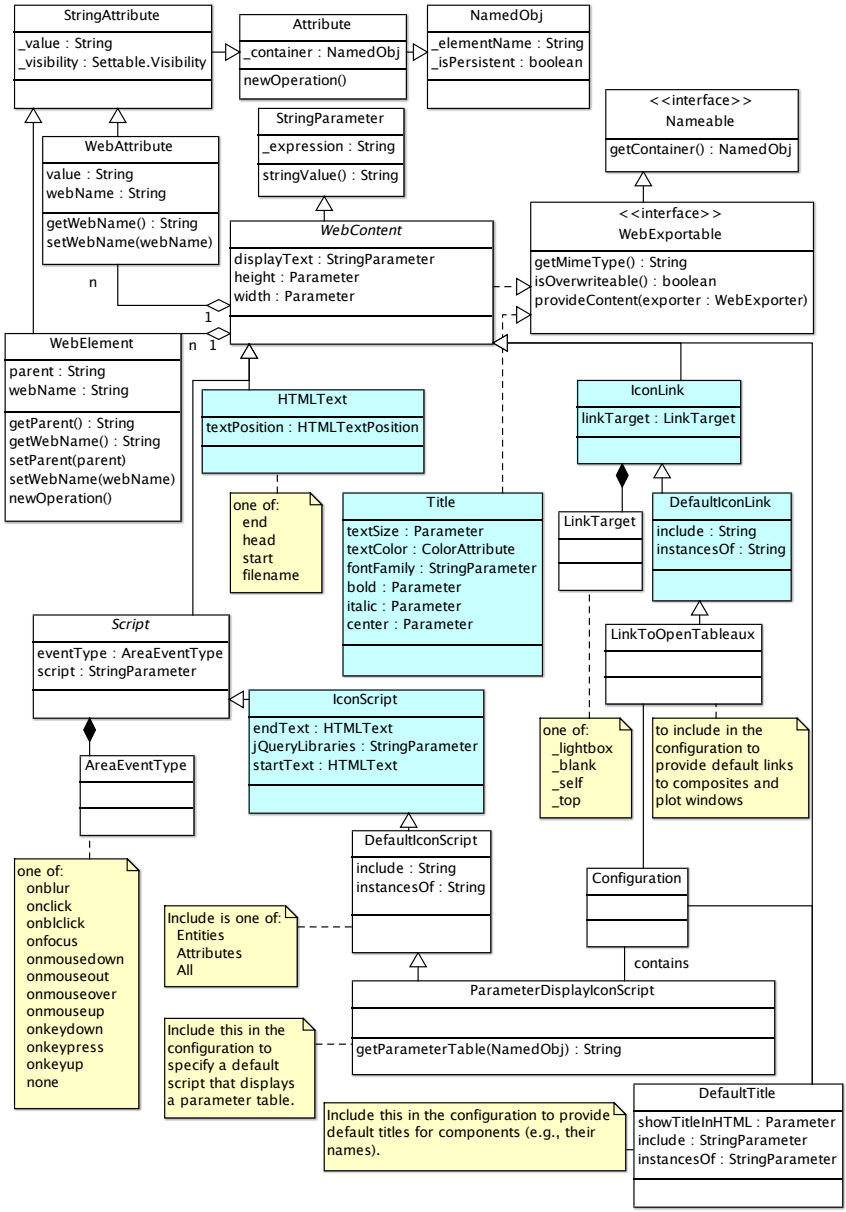


Figure 16.5: UML class diagram for the attributes for customization of exported web pages. The shaded attributes are the most commonly used in models.

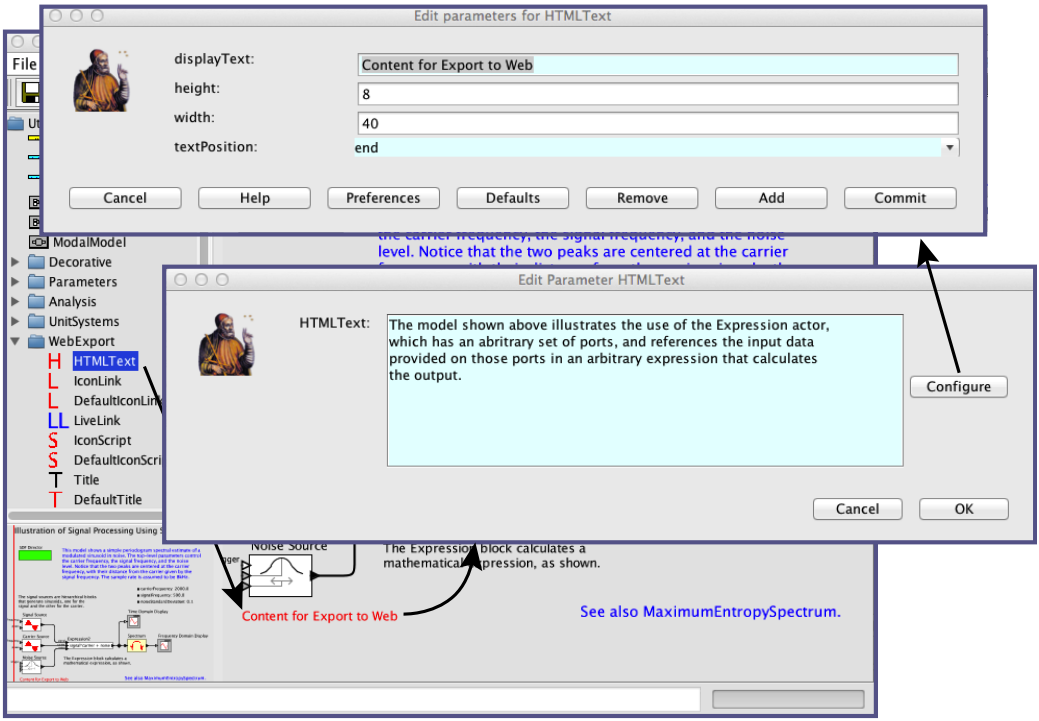


Figure 16.6: Dialog for customizing HTML text to include in an exported web page.

formatting directives. The web page including the text specified in Figure 16.6 is shown in Figure 16.7.

By default, this text will be placed before the image for the model, but you can change the position by setting the *textPosition* parameter, as shown in Figure 16.6. In that figure, you can see that the *HTMLText* attribute is configured to put the text at the end of the HTML file, which explains why that text appears at the bottom of the page in Figure 16.7.

The *HTMLText* attribute has several options for customizing it:

- *displayText*: This parameter determines what shows up in the model itself. By default, this is the text "Content for Export to Web." Notice that this text also appears in the exported web page in Figure 16.7, which is a bit odd. This text is not an interesting part

of the model; it is simply a placeholder for an attribute that customizes the exported web page. If you do not want this attribute to show up in an exported web page, you can simply move the attribute out of the field of view before doing the export. Alternatively, you can set *displayText* to an empty string, but this technique has the disadvantage of making it slightly more difficult to find the attribute to edit or customize the exported text. In Figure 16.8, the *displayText* has been set to the empty string. The *HTMLText* parameter is still present and can be selected (the small yellow box that is barely visible at the lower left in the figure is the *HTMLText* parameter), but since there is no visible icon, it is hard to find. An easier way to edit the *HTMLText* parameter is to right click on the background of the model, as shown in Figure 16.8. The *HTMLText* parameter appears as a parameter of the model, along with whatever other parameters have been defined in the model.

- *height*: The height of the editing box for specifying the text to export. If you change this value, close and re-open the dialog for the change to take effect.
- *width*: The width of the editing box for specifying the text to export. If you change this value, close and re-open the dialog to see the change.
- *textPosition*: As mentioned above, this parameter determines the position of the exported text. The built-in options are end, start, and head. Choosing "end" puts the text after the exported model image. Choosing "start" puts the text before the exported model image. Choosing "head" puts the text in the header section of the HTML page. If you specify any other value for *textPosition*, then that value is assumed to be the name of a file, and a file with that name is created in the same directory as the export. The specified text is then exported to that file.

IconLink: Specifying Hyperlinks for Icons

The *IconLink* parameter shown in the `Utilities→WebExport` library can be used to specify a hyperlink for an icon in the model. To use it, drag it from the library onto the icon that you would like to have the link. In the example of Figure 16.9, we have done such a drag onto the text annotation shown at the lower right that reads "See also MaximumEntropySpectrum." Double clicking on the text annotation reveals an *IconLink* parameter that can be set to a URL. The exported web page will include a hyperlink from the text annotation to that specified page.

The *IconLink* parameter can be customized (click on the Configure button at the lower right of the dialog in Figure 16.9). The parameters *displayText*, *width*, and *height* are the

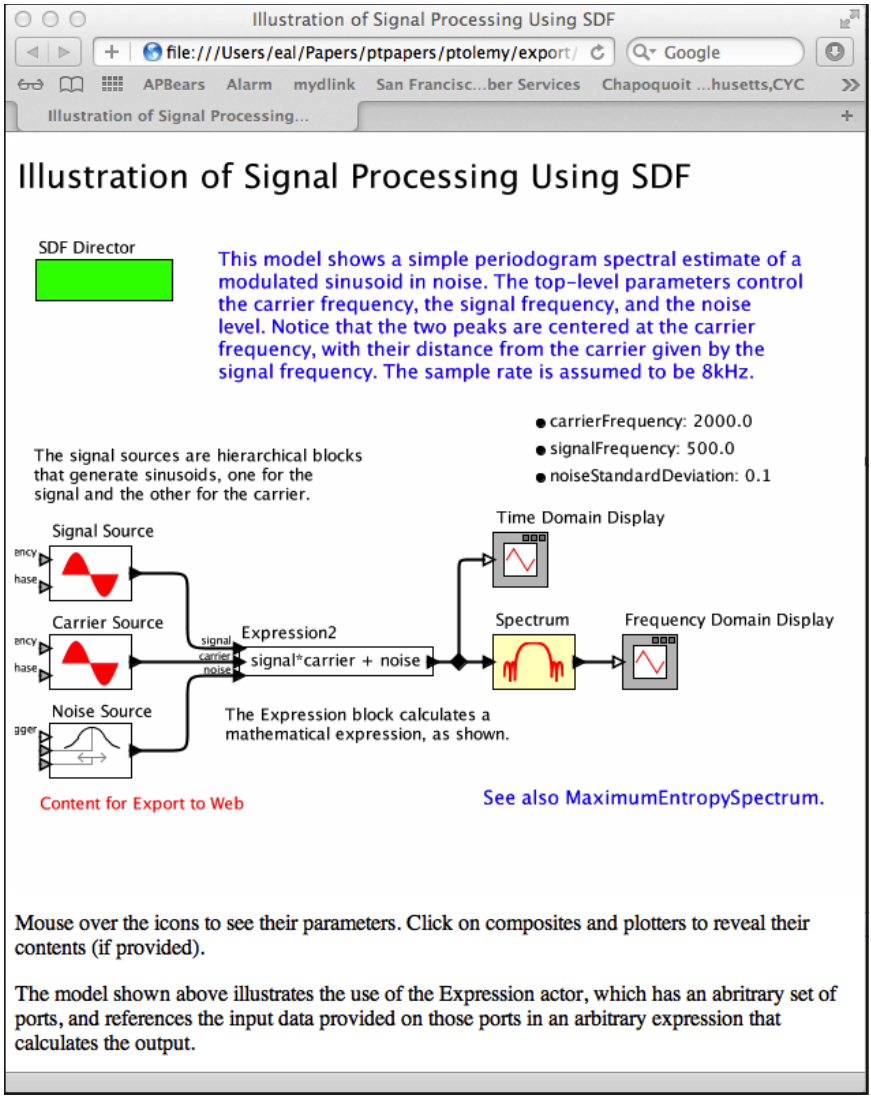


Figure 16.7: Page resulting from inserting an HTMLText attribute into the example of Figure 16.4 and configuring it as shown in Figure 16.6.

same as those for *HTMLText*, described above. A new parameter is *linkTarget*. This has four allowed values:

- `_lightbox`: Display the link in a pop-up lightbox.
- `_blank` (the default): Display the link in new blank window of the browser.
- `_self`: Display the link in the same window, replacing the current page or frame.
- `_top`: Display the link in the same window, replacing the current page.

An example of the lightbox display is the plot shown in Figure 16.3.

In addition, if the *linkTarget* parameter is given any other value, then that value is assumed to be the name of a frame in the web page, and that frame becomes the target.

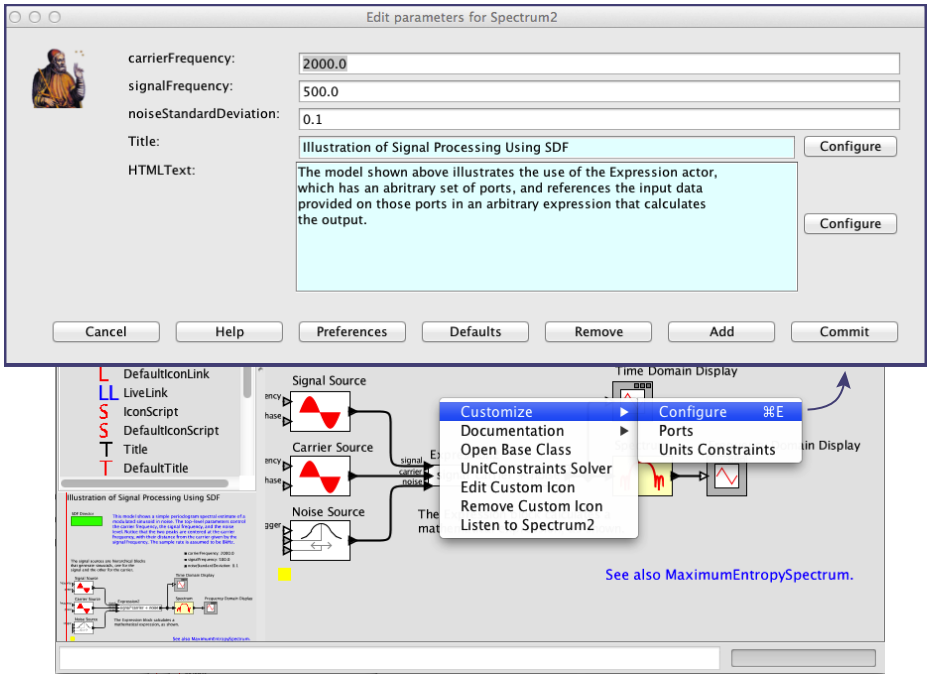


Figure 16.8: The *HTMLText* attribute can be hidden by setting its *displayText* parameter to the empty string. It can still be edited by right clicking on the background of the model. Notice that *HTMLText* appears in the list of model parameters.

DefaultIconLink: Default Hyperlinks for Icons

The *DefaultIconLink* parameter shown in the `Utilities→WebExport` library on the left in Figure 16.4 can be used to specify a default hyperlink for any icon in a model that does not contain an *IconLink*. In addition to the parameters of *IconLink*, *DefaultIconLink* has two additional parameters:

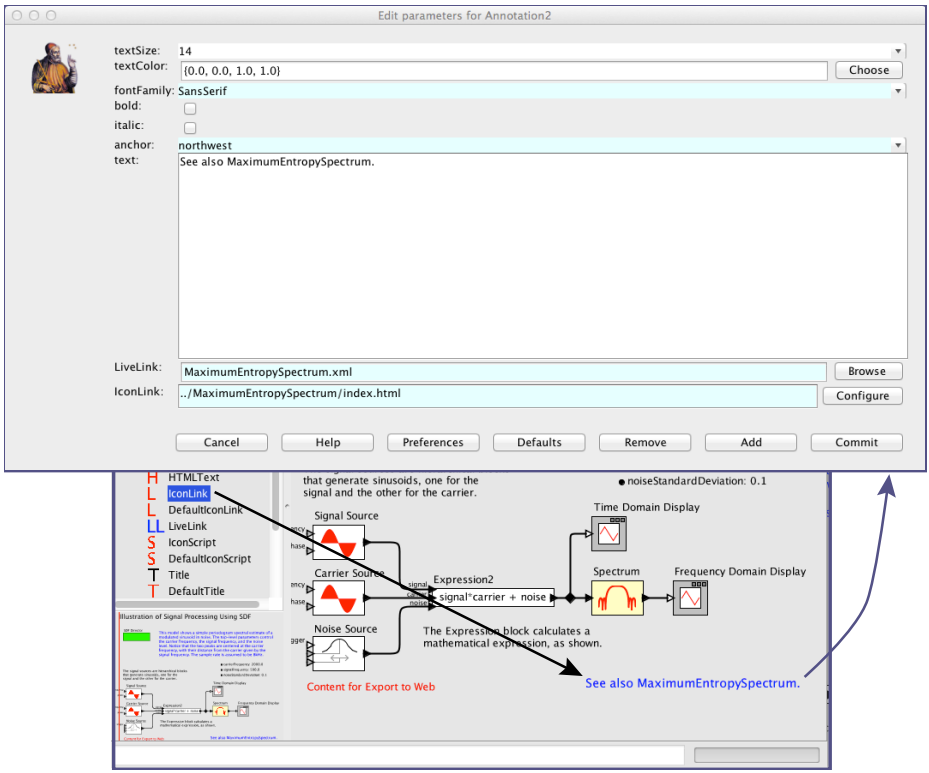


Figure 16.9: The *IconLink* attribute can be dragged onto an icon. The object onto which it is dragged acquires a parameter that can be used to specify a web page to link to from that icon when the model is exported to the web. Here, the exported web page will have a link on this icon to "../../MaximumEntropySpectrum/index.html".

- *include*: This parameter can be used to restrict icons to which the default applies. Specifically, the defaults may be specified for icons for attributes, entities, or both.
- *instancesOf*: If non-empty, this attribute specifies a class name. Only entities or attributes (depending on the *include* parameter) implementing the specified class will be assigned the default link.

LiveLink: Hyperlinks in Vergil

Although not directly related to web page exporting, the *LiveLink* parameter is included in the library because it works particularly well with *IconLink*. In particular, if you drop an instance of *LiveLink* onto an icon, then you can specify a file or URL to be opened when a user double clicks on the icon in Vergil (vs. clicking on an icon in a browser showing the exported web page). This does not automatically result in a hyperlink in an exported web page because typically a model will want to specify a different file or URL to be opened by Vergil than what would be opened by a browser. Vergil can open and display MoML files, for example, whereas a browser will simply display the XML content.

Example 16.1: Notice that in Figure 16.9, the annotation that reads "See also MaximumEntropySpectrum" contains both an instance of *IconLink* and an instance of *LiveLink*. The *LiveLink* references a MoML file, MaximumEntropySpectrum.xml, assumed to be stored in the same directory as the Spectrum model. The *IconLink* parameter, however, references an HTML file. That reference assumes that both Spectrum and MaximumEntropySpectrum will have exported web pages, and that the relative locations of these pages on a server are such that the specified path will provide a link to the HTML file for the MaximumEntropySpectrum.

Assuming all files are arranged appropriately in the file system, the Vergil hyperlink and the web page hyperlink will do essentially the same thing. They will each open the referenced model, MaximumEntropySpectrum. But Vergil will open it in Vergil, whereas a browser will open its exported web page in the browser.

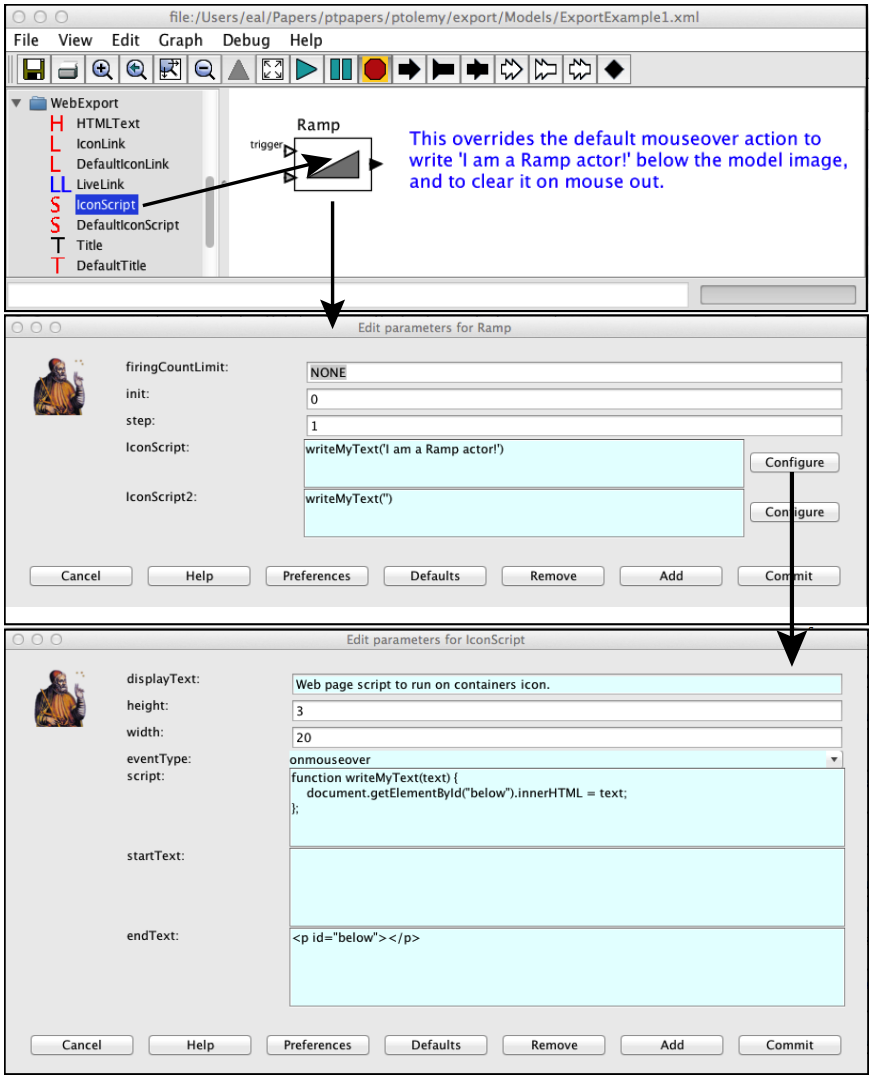


Figure 16.10: Here, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display “I am a Ramp actor!” when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

IconScript: Scripted Actions for Icons

The *IconScript* parameter is used to provide a scripted action associated with an icon in a model. Specifically, an action can be associated with mouse movement over the icon, mouse clicks, or keyboard actions. The action is specified as a JavaScript script.

Example 16.2: An example using *IconScript* is shown in Figures 16.10 and 16.11. In this example, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display "I am a Ramp actor!" when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

The way that this works is that the value of the first *IconScript* parameter is the JavaScript code:

```
writeMyText('I am a Ramp actor!')
```

This invokes a JavaScript procedure `writeMyText`, which is defined in the *script* parameter of the *IconScript* parameter to be:

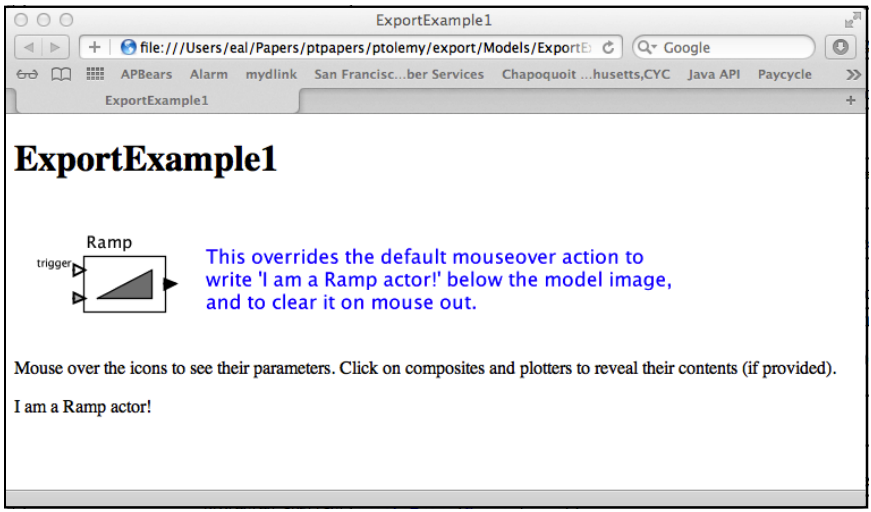


Figure 16.11: Web page exported by the model in Figure 16.11, shown with the mouse lingering over the Ramp icon.

```
function writeMyText(text) {
    document.getElementById("below").innerHTML = text;
};
```

This procedure takes one argument, `text`, and writes the value of this argument into the `innerHTML` field of the element with ID `below`. That element is defined in the *endText* parameter of the *IconScript* parameter as follows:

```
<p id="below"></p>
```

This is an HTML paragraph with ID `below`. This paragraph will be inserted into the exported web page below the model image. Finally, the *eventType* parameter of the *IconScript* is set to `onmouseover`, which results in the script being invoked when the mouse enters the area of the web page displaying the Ramp icon, as shown in Figure 16.11.

The second instance of *IconScript*, named IconScript2, specifies the following script:

```
writeMyText('')
```

This uses the same JavaScript procedure to clear the display when mouse exits the Ramp icon. The *eventType* parameter of this second *IconScript* is set to `onmouseout`.

If multiple instances of *IconScript* have exactly the same *script* parameter, then the value of that parameter will be included only once in the head section of the exported HTML page. Hence, the value of the *script* parameter is required JavaScript definitions. The web page exporter is smart enough to include those definitions only once if they are required at least once in the model.

DefaultIconScript: Default Scripted Actions for Icons

DefaultIconScript is similar to *IconScript*, except that it gets dragged onto the background of a model rather than onto an icon, and it specifies actions for many icons instead of just one. It has the same parameters as *IconScript*, but like *DefaultIconLink* described above,

it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1.

DefaultIconScript can be used, for example, to override the default behavior that causes parameters to be displayed on mouse over, as shown in Figure 16.2.

Title: Title for Icons

The *Title* parameter is used to customize the title displayed in a web page. This parameter also appears as a title in the Vergil window. The title in Figure 16.7 is actually given by an instance of *Title* inserted into the model, with the default title changed to read "Illustration of Signal Processing Using SDF." This replaces the default title provided by the web export, which is the name of the model. This title also becomes the title defined in the header of the exported HTML file.

The default value of the *Title* parameter is the expression

```
$(this.getName())
```

which is an expression in the Ptolemy II expression language for string parameters (see Chapter 13). This expression invokes the `getName` method on the container object, so the default title that is displayed is the name of the model.

DefaultTitle: DefaultTitle for Icons

The *DefaultTitle* parameter is used to customize the title associated with each icon in a model. This title is what shows up on the exported web page as a tooltip when the mouse lingers over an icon. Like *DefaultIconLink* described above, it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1. These can be used to specify default titles for subsets of icons.

16.2 Web Services

Ptolemy allows models to be run as web services. A **web service** runs on a server and is accessible on the Internet via a uniform resource locator (**URL**). Typically, a web service responds to requests by providing either a web page (typically formatted in **HTML**, the

hypertext markup language) or by providing data in some other standard Internet format such as **XML** (the extensible markup language) or **JSON** (the JavaScript object notation). The standard Ptolemy II library includes an attribute that turns a model into a web server, an actor to respond to HTTP requests, actors that facilitate constructing an HTML response, and actors for a model to access and use a web service.

16.2.1 Architecture of a Web Server

Figure 16.12 illustrates the operation of a web server. The URL for accessing the web server consists of the protocol, host name and port number (if other than the default, 80). For example, the URL `http://localhost:8078/` sends an HTTP request to the web server running on the local machine at port 8078.

A web server hosts one or more web applications (or web services). In our case, each application will be realized by one Ptolemy II model containing an instance of the WebServer attribute (see box on page 584). Each application registers an **application path** with the server. The application will handle an HTTP request for URLs that include the application path immediately after the hostname and port. For example, if application 2 registers the application path `/app2`, then the URL `http://localhost:8078/app2` will be handled by application 2. The application path can be the empty string, in which case all HTTP requests to this host on this port will be delegated to the

Sidebar: Command-Line Export

Given a MoML file for a model, you can generate a web page using a command-line program called `ptweb`. The command should have the following form:

```
ptweb [options] model [targetDirectory]
```

The "model" argument should be a MoML file. If no target directory is specified, then the name of the model becomes the name of the target directory (after any special characters have been replaced by characters that are allowed in file names). The options include:

- `-help`: Print a help message.
- `-run`: Run the model before the web page is exported, so that plot windows are included the export.

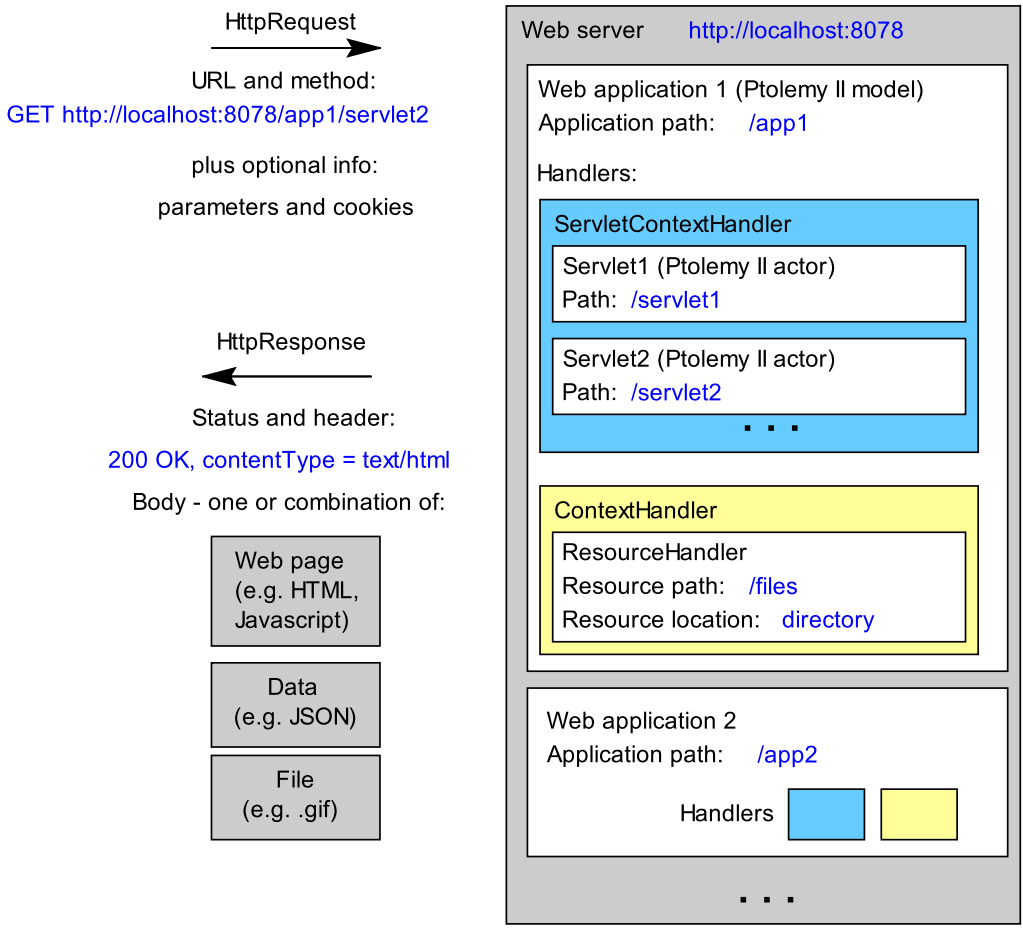


Figure 16.12: A web server hosts one or more web applications. Each application contains one or more request handlers. The web server receives HttpRequests and, according to the URL of the request, delegates the request to the appropriate request handler. The handler returns an HttpResponse.

application. When multiple applications are running on the same server, each application should have a unique application path prefix, so that the server can determine where to delegate requests.

Each application contains one or more request handlers. In our case, these handlers can be instances of the HttpActor actor (see box on page 584). Each handler registers a path prefix with the web application (this path prefix can again be the empty string). For HttpActor, the path prefix is given by the *path* parameter. When multiple handlers are running in the same application, each should have a unique path prefix, so that the application can determine where to delegate the request. For example, in Figure 16.12, the URL `http://localhost:8078/app1/servlet2` will be handled by application 1, which will delegate it to a Ptolemy II actor that has registered the prefix `servlet2`. If more that one prefix matches, then the server will delegate to handler with the most specific prefix. For example, if one handler has a blank prefix and the other has the prefix `/foo`, then all requests of the form `http://hostname:port/applicationPath/foo/...` will be delegated to the second handler, and all other requests to the first.

A second type of handler called a **resource handler** is also provided by the WebServer to handle requests for static resources such as files (Jetty class `ResourceHandler`). Again, this has a prefix which must appear in the URL. For example, in Figure 16.12, the URL `http://localhost:8078/app1/files/foo.png` references a file named `foo.png` that is stored on the server in a directory identified by a resource location attribute of the WebServer.

The response produced by a handler contains a status code, header, and the response body. The response body is the content for the user, for example, a web page, a file, or data formatted in JSON. The status code indicates whether the operation was successful, and if not, why not. There is a standard set of response codes for HTTP requests[†] The header contains information such as the content format (the **MIME type**[‡], the content length, and other useful information.

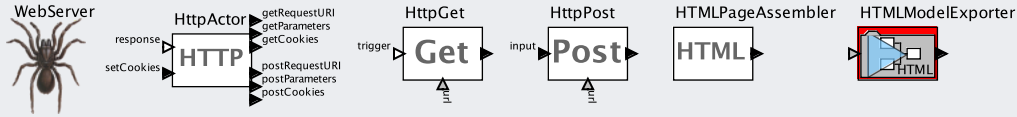
16.2.2 Constructing Web Services

The use of the WebServer and HttpActor are illustrated by the following example.

[†]See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.
[‡]See `http://www.iana.org/assignments/media-types/index.html`.

Sidebar: Components for Web Services and Web Pages

Some components that are particularly useful for constructing web services, accessing web pages, and building web pages are shown below:



- **WebServer**. An attribute that starts a Jetty web server (see `http://www.eclipse.org/jetty/`) when the model containing it is executed. This attribute routes incoming HTTP requests to objects in the model that implement an HttpService interface, such as HttpActor. This attribute has parameters for specifying the port on which to receive HTTP requests, an application path to be included in the URL accessing this server, directories in which to find resources that are requested, and a directory in which to store temporary files.

- **HttpActor**. An actor that handles **HTTP GET** and **HTTP POST** requests that match its path. This actor is designed to work with the DE director. The outputs contain the details of the request time stamped by the elapsed time (in seconds) since the server model started executing. This actor expects that for each output it produces, the model in which it resides will provide an input that is the response to the HTTP request.

- **HttpGet**. An actor that issues an HTTP GET request to a specified URL. This actor is similar to FileReader, but it only handles URLs, and not files.

- **HttpPost**. An actor that issues an HTTP POST request to a specified URL. The contents of the post are specified by an input record.

- **HTMLPageAssembler**. An actor that assembles an HTML page by inserting input text at appropriate places in a template file.

- **HTMLModelExporter**. An extension of the VisualModelReference actor that not only displays and executes a referenced model, but also exports that model to a web page using the techniques discussed in Section 16.1.

Example 16.3: The model in Figure 16.13 is a web service that asks the user to type in some text, then returns the "Ptolemnized" text, where all leading 'p's (but not including instances of 'th') are replaced with 'pt'. For example, 'text' becomes 'ptext', as shown in Figure 16.14. The text manipulation is accomplished by the PythonScript actor, which executes the Python code shown in Figure 16.15.

First, notice that the *stopWhenQueueIsEmpty* parameter of the DE director is set to false. Were this not the case, the model would halt immediately when run because there would be no pending events to process. Second, notice that the *enable-*

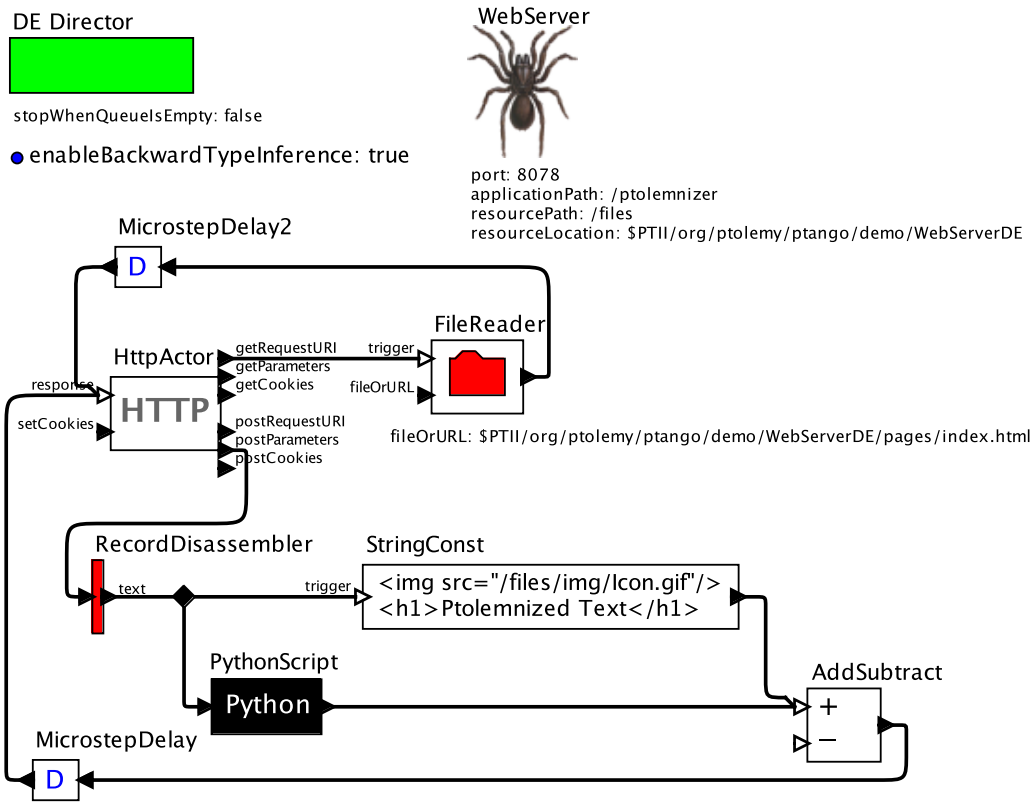


Figure 16.13: A simple web service implemented in Ptolemy II. [online]

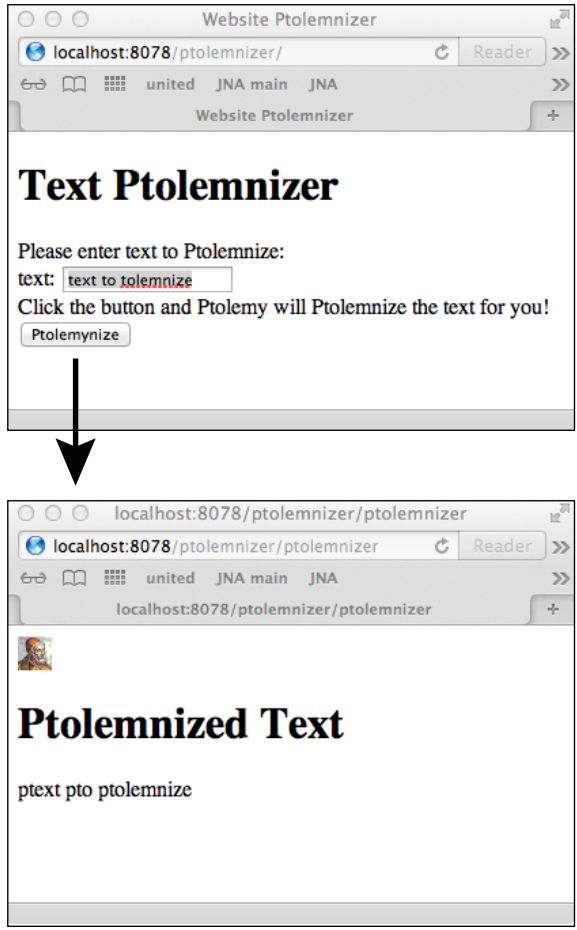


Figure 16.14: The web page returned by model in Figure 16.13 in response to an HTTP GET, and the page returned in response to a POST (triggered by the button).

```
from ptolemy.data import StringToken
class Main :
  "ptolemizer"
  def fire(self) :
    # read input, compute, send output
    t = self.in.get(0)
    s = t.stringValue()
    s = self.ptolemize(s)
    t = StringToken(s)
    self.out.broadcast(t)
    return

  def ptolemize(self, s) :
    l = list(s)
    length = len(l)
    if length == 0 :
      return ''
    if length == 1 :
      if l[0] == 't' :
        return 'pt'
      else :
        return l[0]
    if l[0] == 't' and l[1] != 'h' :
      l[0] = 'pt'
    i = 1
    while i < length - 1 :
      if l[i-1] == ' ' and l[i] == 't' and l[i+1] != 'h' :
        l[i] = 'pt'
      i = i + 1
    if l[-2] == ' ' and l[-1] == 't' :
      l[-1] = 'pt'
    return reduce(lambda x,y: x+y, l, '')
```

Figure 16.15: The Python code for the PythonScript actor in Figure 16.13.

BackwardTypeInference parameter of the model is set to true. This enables backward type inference, which in this case, results in the *postParameters* output of the HttpActor to have a record type with a single field called "test" of type string. The PythonScript actor specifies the type *string* on its input port, because the Python code expects a string.

When this model is executed, the WebServer launches a web service with an application path of `/ptolemnizer` on port 8078 of the local machine. The service is therefore available at `http://localhost:8078/ptolemnizer`. Accessing that URL in a web browser results in the top web page of Figure 16.14. How does this work?

When the web server receives an HTTP GET request with a matching application path, it delegates the request to the HttpActor. The actor requests of the director to be fired, and when the director fires it, it produces information about the GET request on its top three output ports. This model uses the URL of the GET request to trigger the FileReader actor, which simply reads a file on the local file system, the contents of which are shown in Figure 16.16. The contents of that file are sent back to the *response* input of the HttpActor, which then fires again. On that second firing, it collaborates with the WebServer to serve the response shown at the top of Figure 16.14. Note that MicrostepDelay actor is required in the feedback loop, as usual for DE models (see Section 7.3.2).

As you can see in Figure 16.14 and Figure 16.16, the web page that is served has a form, and pushing the "Ptolemnize" button results in an HTTP POST with the contents of the form. When this POST occurs, the WebServer again delegates to the HttpActor, which outputs the details of the POST on its lower three output ports. The *postParameters* port will produce a record token with a single field called "text." The RecordDisassembler extracts the value of this field, which is the text entered by the user into the form. The StringConst, PythonScript, and AddSubtract actor then construct an HTML response, which is sent back to the HttpActor. That response results in the page at the bottom of Figure 16.14.

The response to the POST includes an "img" element (see the StringConst actor in Figure 16.13). When the browser parses this response, this img element will trigger another HTTP GET. The WebServer has its *resourcePath* parameter set to `/files`, so the img src URL `/files/img/Icon.gif` will be handled by the resource handler rather than being delegated to an HttpActor (see Figure 16.12). That resource handler will search for a file named `img/Icon.gif` in the directory

given by the *resourceLocation* parameter. The small Ptolemy icon on the bottom page of Figure 16.14 is the result.

This example constructs a web service by composing a number of capabilities. It uses HTML to construct an interactive web page, and Python to process data submitted by a user. In effect, the Ptolemy model is serving as an orchestrator for a number of distinct software components.

16.2.3 Storing Data on the Client using Cookies

A **cookie** is small piece of data — specifically a (name, value) pair plus expiration and visibility information — that is stored by a web browser on the client side and returned to the web server along with subsequent HTTP requests. A web service can store state on the client using a cookie; for example, a web service can use a cookie to remember that the user has logged in. A **persistent cookie** is stored for a specified period of time (including indefinitely), whereas a **session cookie** is only stored until the browser window is closed.

HttpActor has basic support for getting and setting session cookies from a client browser. Specifically, HttpActor has a *requestedCookies* parameter whose value is an array of strings. This specifies the names of cookies that the web service sets or gets. It also has an input port *setCookies*, which accepts a record that assigns values to each of the named cookies. Finally, it has output ports *getCookies* and *postCookies* that provide a record with cookie values along with each HTTP GET or POST request.

Example 16.4: The model shown in Figure 16.17 uses cookies. The web service that this model implements uses cookies to remember the identity of a client over a sequence of HTTP accesses. The pages shown in Figure 16.18 illustrate how the service responds to an initial HTTP GET, an HTTP POST that stores the identity of a client "Claudius Ptolemaeus" as a cookie, a subsequent HTTP GET, and finally, an HTTP POST that deletes the cookie.

The model has two instances of HttpActor. The first one, labeled HttpActor1, has the default *path* parameter, which matches all requests. The second one, labeled

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title> Website Ptolemnizer </title>
</head>
<body>

<div data-role="page" data-theme="c">
  <div data-role="header">
    <h1> Text Ptolemnizer </h1>
  </div>
  <div data-role="content">
    Please enter text to Ptolemnize:
    <form action="ptolemnizer" method="post" >

      <div data-role="fieldcontain" class="ui-hide-label">
        <label for="text">text:</label>
        <input type="text" name="text" id="text" value=""
                  width="80" placeholder="text to tolemnize"/>
        </br>
      </div>

      <div>
        Click the button and Ptolemy will
        Ptolemnize the text for you!
        <br/>
        <button type="submit" id="ptolemnize">
          Ptolemynize
        </button>
      </div>
    </form>
  </div>
</div>
</body>
</html>
```

Figure 16.16: The HTML code read by the FileReader actor in Figure 16.13.

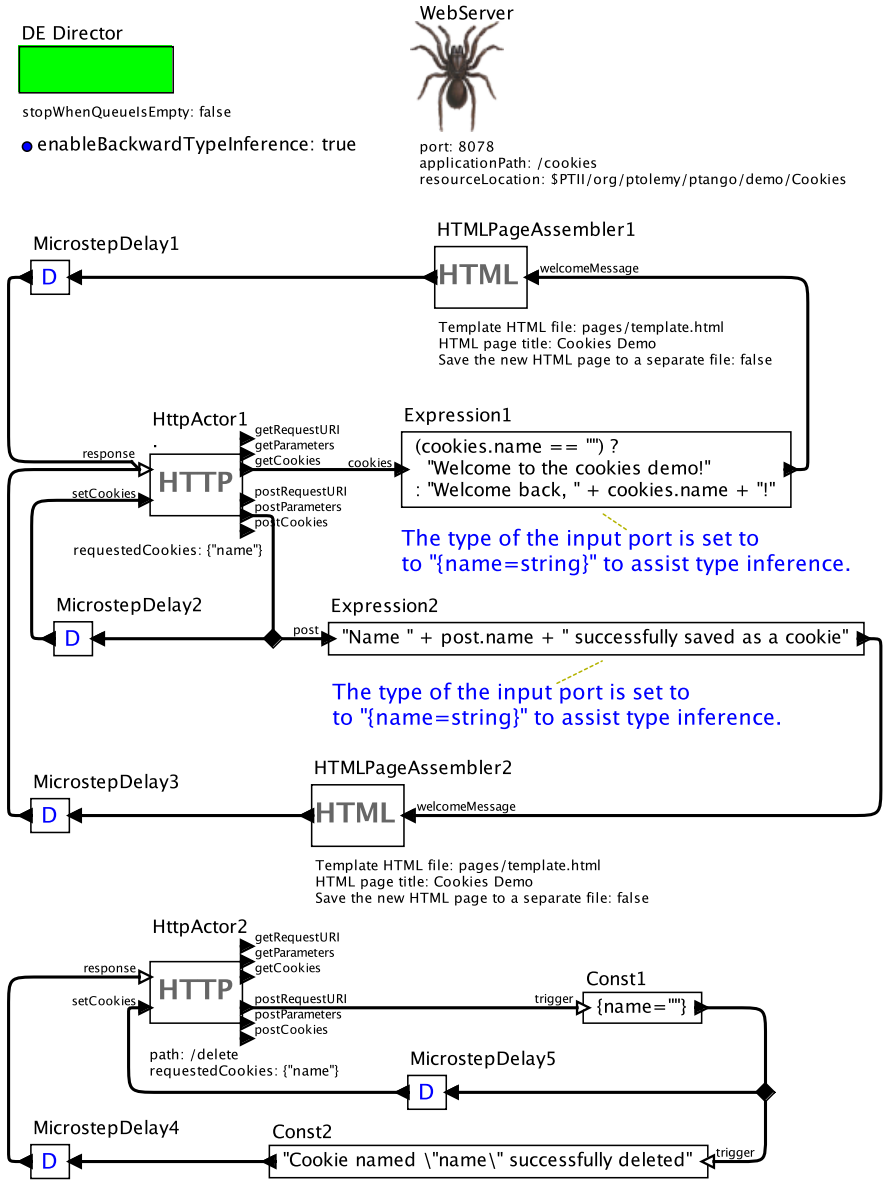


Figure 16.17: A model that gets, sets, and deletes a cookie on the client. [online]

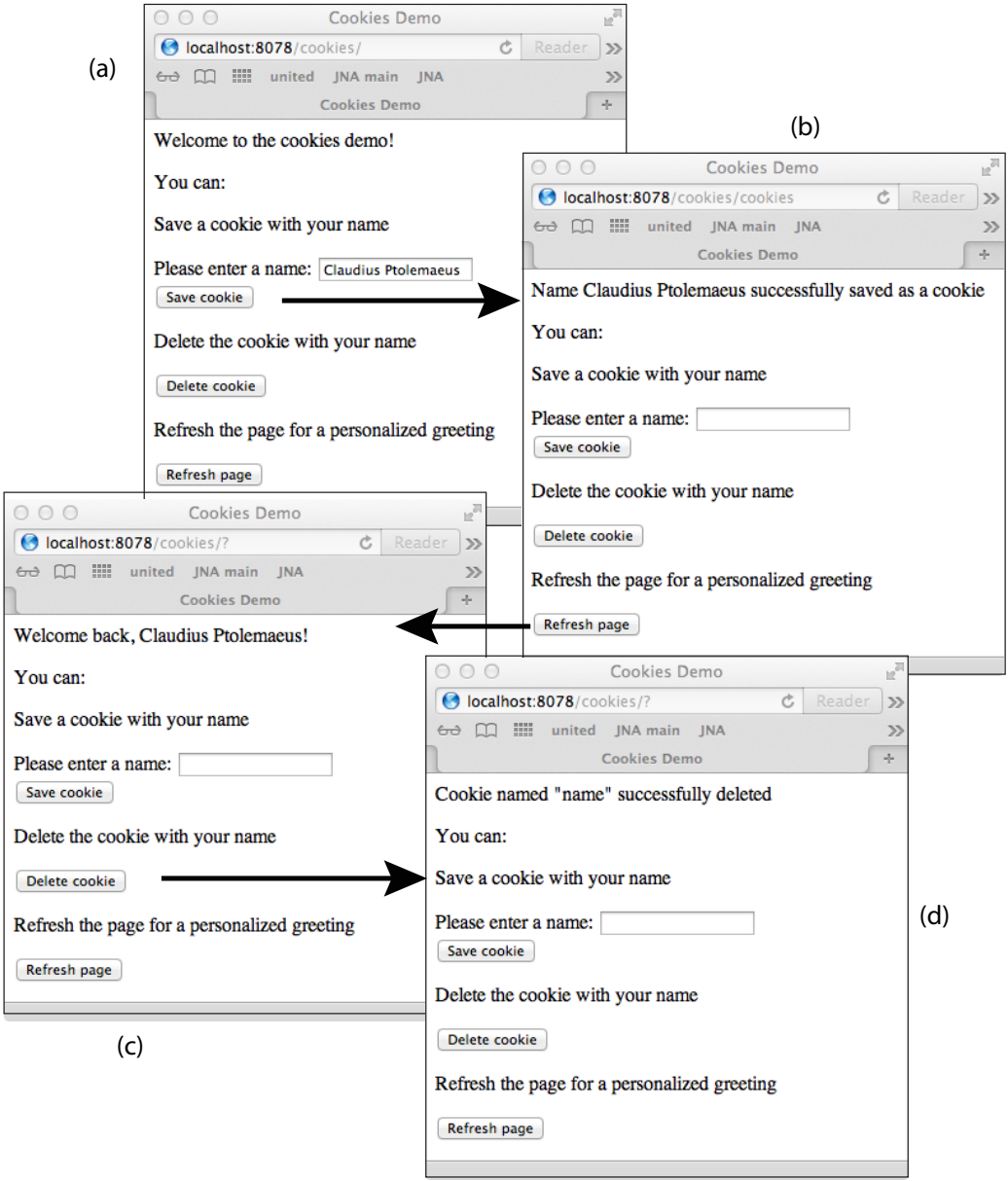


Figure 16.18: A sequence of web pages created by the model in Figure 16.17.

HttpActor2, has *path* set to `/delete`, so it will handle requests with URLs of the form `http://localhost:8078/cookies/delete`.

Both instances of HttpActor have parameter *requestedCookies* set to `{"name"}`, and array with one string. This instructs the HttpActor to check the incoming HTTP request for a cookie with the label `name`. The HttpActor produces a record on its *getCookies* or *postCookies* output port with the label `name` and the value provided by the cookie. If no cookie is found, the value is an empty string.

Note that an HttpActor actor always produces a record with the fields specified in *requestedCookies*, so downstream actors can always assume a record with the specified field. Hence, for example, the Expression actor named Expression1 in Figure 16.17 extracts the `name` field of the record using the syntax `cookies.name`. If value of the field is an empty string, then the model generates a generic welcome message, as shown in Figure 16.18(a). Otherwise, it customizes the page, as shown in Figure 16.18(c).

From the initial page, Figure 16.18(a), the user can specify a name and save a cookie with the name, which yields the response Figure 16.18(b). This is accomplished using an HTTP POST with parameter `name`. Notice in Figure 16.17 that the *postParameters* output port is fed back to the *setCookies* input port, so the response to this HTTP POST will be to set a cookie in the browser with whatever value is provided by the POST.

Clicking on the "Refresh page" button causes another HTTP GET, which now yields the customized page, Figure 16.18(c).

Clicking on the "Delete cookie" button sends a POST request to `http://localhost:8078/cookies/delete`. This request is mapped to HttpActor2. The response has two parts. First, Const1 sends a record with the label `name` and an empty string value to the *setCookies* port on HttpActor2. HttpActor2 interprets this as a request to delete the cookie. Note that, because of this implementation, the HttpActor actor will interpret any `RecordToken` label with an empty string value as a request to delete the cookie with that label. Hence, a missing cookie is equivalent to a cookie with an empty value. In addition, the model will generate a response confirming deletion of the cookie, Figure 16.18(d).

Assembling Web Pages

The model in Example 16.4 and Figure 16.17 serves some non-trivial web pages. To facilitate construction of these web pages, the model uses the HTMLPageAssembler actor. This actor inserts contents from its input ports into a specified template file, and outputs the resulting HTML page. The names of the input ports match HTML tag IDs in the template file.

Example 16.5: Figure 16.19 shows the HTML template referenced by the HTMLPageAssembler actors in Figure 16.17. Notice the **div** tag with ID "welcomeMessage." Notice further that the actors each have an input port named *welcomeMessage*, which has been added by the builder of the model. Whatever is received on this input port will be inserted into this div tag position in the response HTML page.

Note that the `Save cookie` and `Refresh page` buttons are HTML forms. These buttons perform the action specified when clicked. For example, the `Save cookie` button generates a POST request to the relative URL `cookies`, at `http://localhost:8078/cookies`, as specified by line 7. The `Refresh page` button generates a GET request to that same URL, as specified by line 24.

An alternative technique, also used in Figure 16.17, is to use JavaScript to update a page instead of returning a new page. This technique is known as **AJAX** (for asynchronous JavaScript and XML).

Example 16.6: The `Delete cookie` button calls the JavaScript function `deleteCookie()`, as shown on lines 17-18 of Figure 16.19. Figure 16.20 shows the `deleteCookie()` function definition. The function submits a POST request to the relative URL `cookies/delete`. If the request is successful, the response data are inserted into the HTML element with the ID `welcomeMessage` (overwriting any previous data). If the request is not successful, an error message is inserted into this element.

This example illustrates two reasons for using Ajax. First, returning a whole page is not necessary for the delete case. A simple message is sufficient. There are many cases

where a developer might want to insert a small update into a larger page. This promotes separation of concerns, where one developer could be responsible for the main page, and a second could be responsible for updates without having to know the structure of the rest of the main page. The second developer might also want to create a web service to provide data to many different pages.

```
    <body>
    <div>
        <div id="welcomeMessage">
        </div>

        <div> <p> You can: </p>  </div>
        <form accept-charset="UTF-8" action="cookies
                      method="post">
            <p> Save a cookie with your name </p>
            <p> Please enter a name:
                <input type="text" name="name" id="name"/>
                <br>
                <input type="submit" value="Save cookie"/>
            </p>
        </form>

        <div> <p> Delete the cookie with your name </p>
           <input type="button" value="Delete cookie"
            onclick="deleteCookie()"/>
        </div>

        <div> <p>
             Refresh the page for a personalized greeting
        </p> </div>

        <form name="input" action="/cookies" method="get">
            <input type="submit" value="Refresh page" />
        </form>

    </div>
    </body>
</html>
```

Figure 16.19: The HTML template referenced by the HTMLPageAssembler actors in Figure 16.17.

```
<!DOCTYPE HTML>
<html>
    <head>
        <script type="text/javascript"
          src="http://code.jquery.com/jquery-1.6.4.min.js">
        </script>
        <script type="text/javascript">
            function deleteCookie() {
                jQuery.ajax({
                      url: "/cookies/delete",
                      type: "post",
                      success: function(data) {
                          jQuery('#welcomeMessage')
                            .html(data);
                          },
                      error: function(data) {
                          jQuery('#welcomeMessage')
                            .html("Error deleting cookie.");
                      }
                    });
            }
        </script>
        <title>Cookies demo</title>

    </head>
```

Figure 16.20: The head section of the HTML template page used in Figure 16.17.

A more subtle reason for using Ajax is that the URL of the website remains unchanged, at `http://localhost:8078/cookies`, while still being able to use a URL structure for the delete web service, `cookies/delete`. If the URL were to change to `http://localhost:8078/cookies/delete`, this would cause problems when the user clicks on further buttons, because the button URLs are defined as relative URLs. E.g., the URL would then be `http://localhost:8078/cookies/delete/cookies`.

There are, of course, many other ways to create web pages to respond to HTTP requests. A particularly interesting possibility is to use the techniques covered in Section 16.1 above to generate web pages from Ptolemy II models. In fact, a web service model could include an instance of the HTMLModelExporter actor, which refers to another Ptolemy II model, executes it, generates a web page with the results, and returns the web page. This offers a particularly powerful way to combine models to provide sophisticated services.

16.3 Summary

Building web pages and web services by constructing models offers a potentially very powerful way to combine sophisticated components in a modular way. At a minimum, the ability to export a web page that documents a model is valuable, enabling teams of designers to more effectively communicate with one another. But more interestingly, the ability to incorporate web servers into models offers a particularly powerful way to combine distributed services.

Exercises

1. Figure 4.3, discussed in Example 4.3 of Chapter 4, implements a simple chat client that uses HTTP Get and HTTP Post to enable a client to chat with other clients on the Internet. In this exercise, we build a simple (and rather limited) web server that supports this client. This server will support exactly two clients, one that will use URL
   `http://localhost:8078/chat/Claudius`
   for its Get requests, and one that will use URL
   `http://localhost:8078/chat/Ptolemaeus`
   for its Get requests. Both will use the same URL,
   `http://localhost:8078/chat/post`
   to post chat data.

   (a) A key property of this server is that it must implement long polling, where it sits on an HTTP Get request until a chat client issues an HTTP Post, which provides some chat text, and then it responds to all clients that have pending Gets with the contents of the Post. To support this, create an actor-oriented class composite actor with two input ports, *get* and *post*, and one output port *response*. This class should queue a get request (at most one) and when a post arrives, if there is a pending get request in the queue, then it should respond with the contents of the post.

   (b) Use the class definition created in part (a) to build a web server that supports the two clients.

   (c) A limitation of the chat client in Figure 4.3 is that it does not stop gracefully. The stop button in the Vergil window eventually stops it, but not until the FileReader actor times out, which can take a long time. In a better design, the server would always respond to an HTTP Get request within some amount of time, given by parameter *maximumResponseTime*. It could respond with an empty string, and the client could then filter out empty strings so that it does not display them to the user. In this design, stopping the client will succeed within the *maximumResponseTime*. Modify your server and the client to implement this.

   (d) (Open-ended question) One of the limitations of the web server you have been asked to design is that only exactly two clients are supported. Another is that there is no authentication of clients. Discuss how to address these limitations,

and implement a more elaborate server that addresses at least one of these limitations.

17 Signal Display

Christopher Brooks and Edward A. Lee

Contents

Ptolemy II includes a number of **signal plotters**, shown in Figure 17.1. These can be found in the `Sinks` library, as shown in Figure 17.2. This appendix gives an overview of these capabilities, with emphasis on how to customize the plots. Once plots have been customized, saving the model containing the plotter will make the customization persistent.

17.1 Overview of Available Plotters

The plotters shown in Figure 17.1 provide a number of capabilities, and are all built on a common infrastructure. The most basic is **SequencePlotter**, which simply plots data values received on the input port. An example of a plot is shown in Figure 17.3. The mechanisms for customizing the title, axes, legend and signal plots will be covered in the following section.

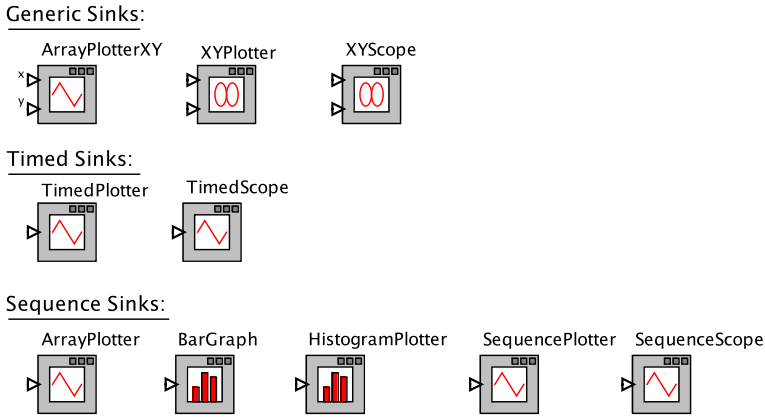


Figure 17.1: Available signal plotters.

The actors shown in the libraries in Figure 17.1 provide a variety of ways of displaying data. **ArrayPlotter** is similar except that it operates on input arrays rather than sequences. Whereas the SequencePlotter actor plots all of the input data over the entire run of the the model, the SequenceScope actor plots windows of input data, optionally overlaying them, as shown in Figure 17.4. The **SequenceScope** actor is more useful with long or infinite runs. It functions more like an oscilloscope in that it forgets old data, plotting only recent data, and overlaying windows of the data on each other.

The **TimedPlotter** actor plots input data as a function of the time stamps of the inputs, as shown in Figure 17.5. This plotter is useful in domains that advance model time, such as DE and Continuous. **TimedScope** is similar, though like SequenceScope, it functions like an oscilloscope and forgets old data.

The **XYPlotter** actor plots input data from one input port vs. input data from its other input port, as shown in Figure 17.6. **XYScope** is similar, though like SequenceScope, it functions like an oscilloscope and forgets old data. **ArrayPlotterXY** is similar, except that it operates on arrays of data rather than sequences.

BarGraph plots input arrays in the form of a bar graph. **HistogramPlotter** calculates a histogram of input data and then plots it as a bar graph, as shown in Figure 17.7. As shown in Figure 17.2, there is also a **ComputeHistogram** actor which calculates a histogram without plotting it.

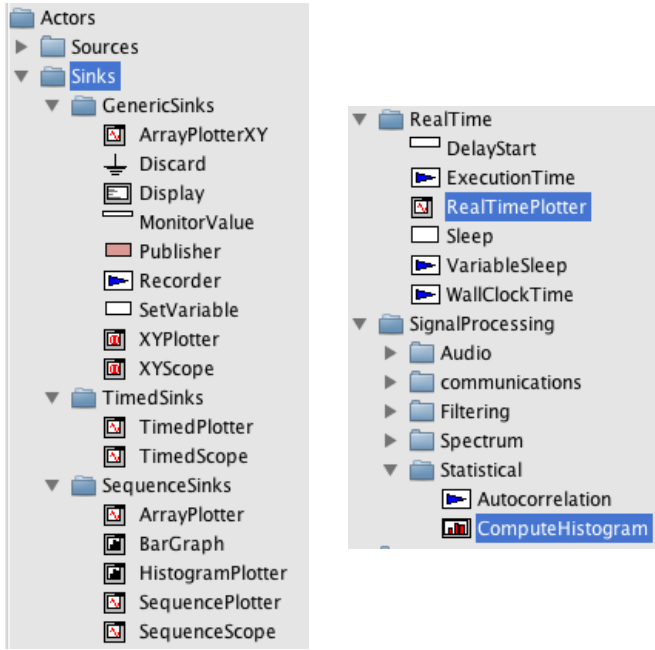


Figure 17.2: A variety of signal plotters can be found in the `Sinks` library.

Also shown in Figure 17.2 is **RealTimePlotter**, which plots input values as a function of real time elapsed on the computer executing the model.

17.2 Customizing a Plot

When used with its default configuration, the SequencePlotter will produce a plot like that shown in Figure 17.8, which is also shown in Figure 3.1. The default title is uninformative, the axes are not labeled, and the horizontal axis ranges from 0 to 255, which is not meaningful.* In the model that created this plot, which is shown in Figure 3.1, in one iteration, the Spectrum actor produces 256 output tokens. By default, the SequencePlotter just numbers these samples 0 to 255, and uses those numbers as the horizontal axis for the plot. But the horizontal axis may have more meaning in the model. In this particu-

*Hint: Notice the "$x10^2$" at the bottom right, which indicates that the label "2.5" stands for "250".

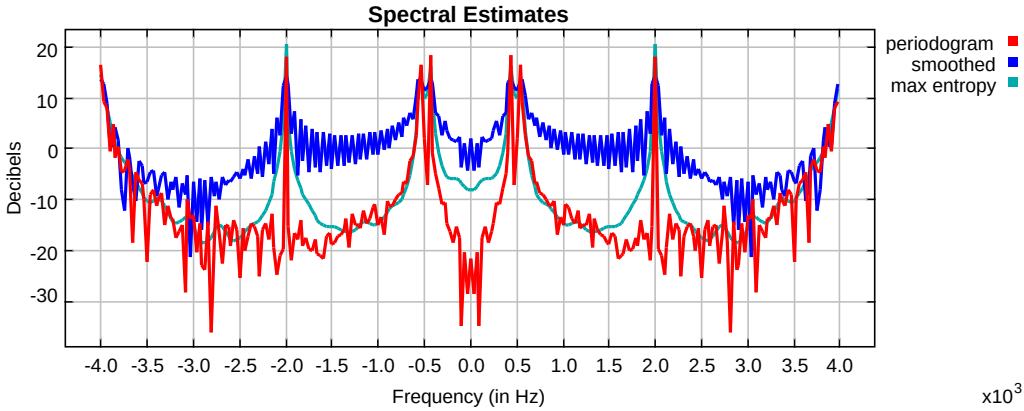


Figure 17.3: Example of a plot produced by the SequencePlotter actor.

lar example plot, the plotted data represent frequency bins that range between $-\pi$ and π radians per second.

The SequencePlotter actor has some pertinent parameters, shown in Figure 17.9, that can be used to improve the labeling of the plot. The *xInit* parameter specifies the value to use on the horizontal axis for the first token. The *xUnit* parameter specifies the value to increment this by for each subsequent token. Setting these to "-PI" and "PI/128" respectively results in the plot shown in Figure 17.10.

This plot is better, but still missing useful information. To control more precisely the visual appearance of the plot, click on the second button from the right in the row of buttons at the top right of the plot. This button brings up a format control window. It is shown in Figure 17.11, filled in with values that result in the plot shown in Figure 17.12. Most of these are self-explanatory, but the following pointers may be useful:

- The grid is turned off to reduce clutter.
- Titles and axis labels have been added.
- The X range and Y range are determined by the fill button at the upper right of the plot.
- Stem plots can be had by clicking on "Stems"
- Individual tokens can be shown by clicking on "dots"
- Connecting lines can be eliminated by deselecting "connect"

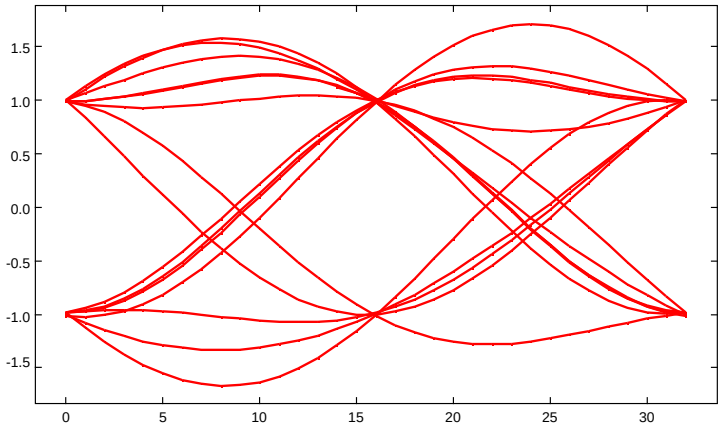


Figure 17.4: Example of a plot produced by the SequenceScope actor, which overlays successive windows of data, like what an oscilloscope does.

- The X axis label has been changed to symbolically indicate multiples of PI/2. This is done by entering the following in the X Ticks field:

  ```
  -PI -3.14159, -PI/2 -1.570795, 0 0.0, PI/2 1.570795, PI 3.14159
  ```

  The syntax in general is: *label value, label value, ...*, where the label is any string (enclosed in quotation marks if it includes spaces), and the value is a number.

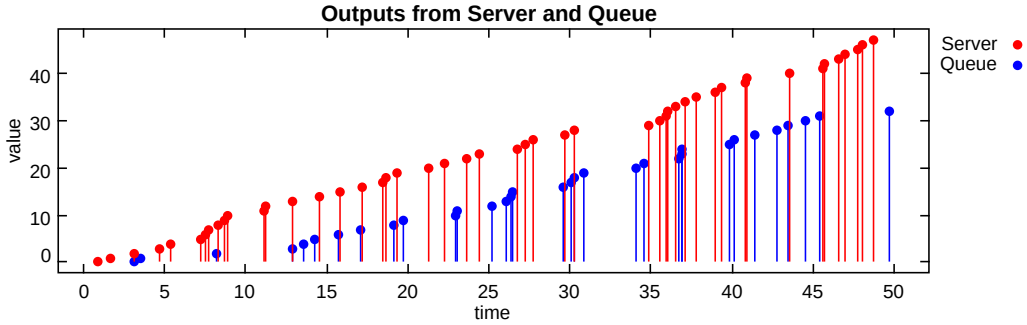


Figure 17.5: Example of a plot produced by the TimedPlotter actor, which plots input data as a function of the time stamps of the inputs.

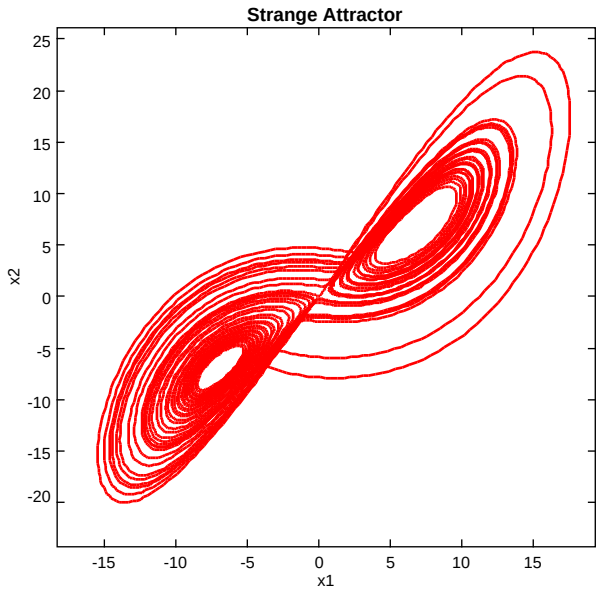


Figure 17.6: Example of a plot produced by the XYPlotter actor, which plots input data on one input port vs. input data on the other port.

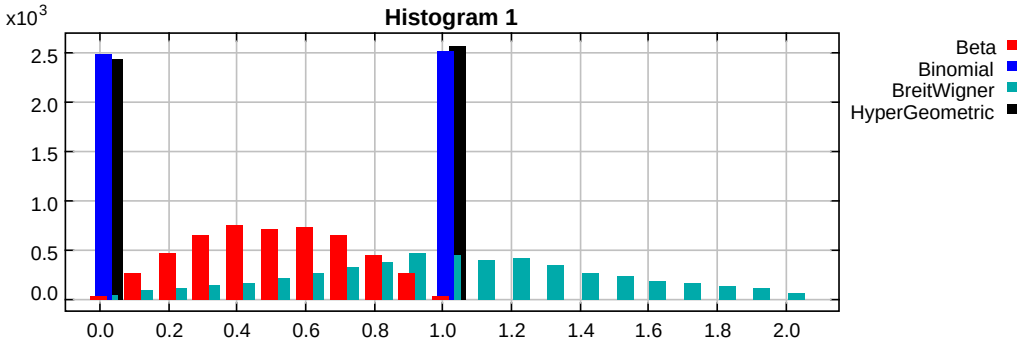


Figure 17.7: Example of a plot produced by the HistogramPlotter actor, which calculates and plots a histogram based on input data.

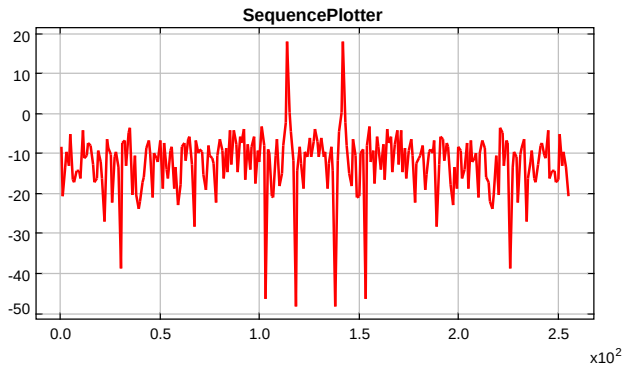


Figure 17.8: In its default configuration, a plot produced by SequencePlotter does not have informative labels.

Figure 17.9: Parameters for the SequencePlotter actor.

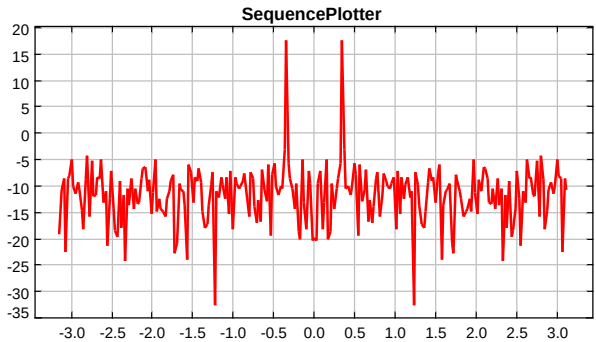


Figure 17.10: Better labeled plot, where the horizontal axis now properly represents the frequency values.

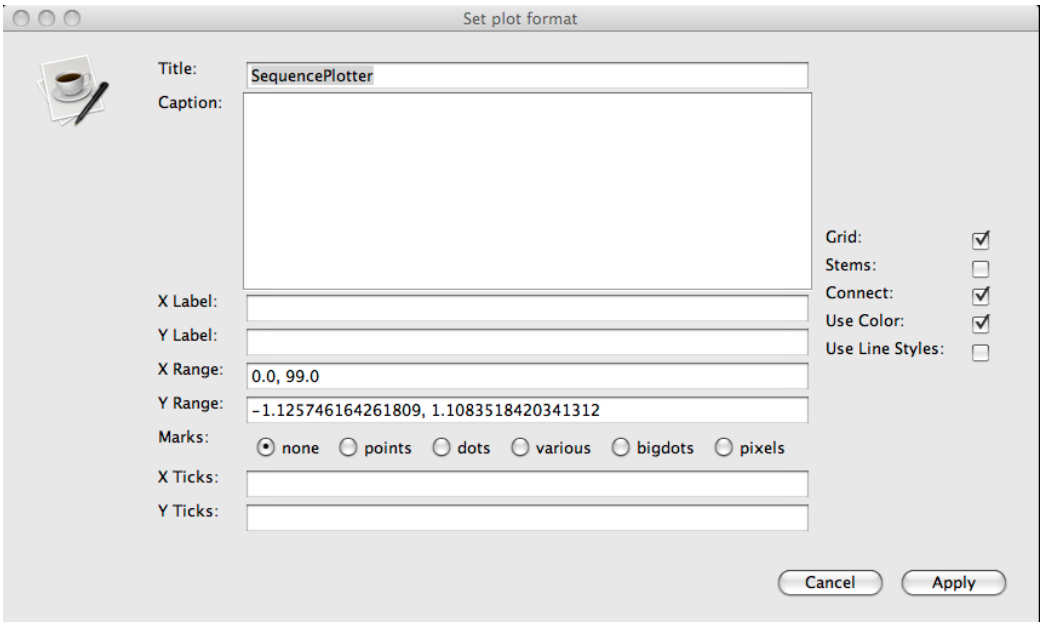


Figure 17.11: Format control window for a plot.

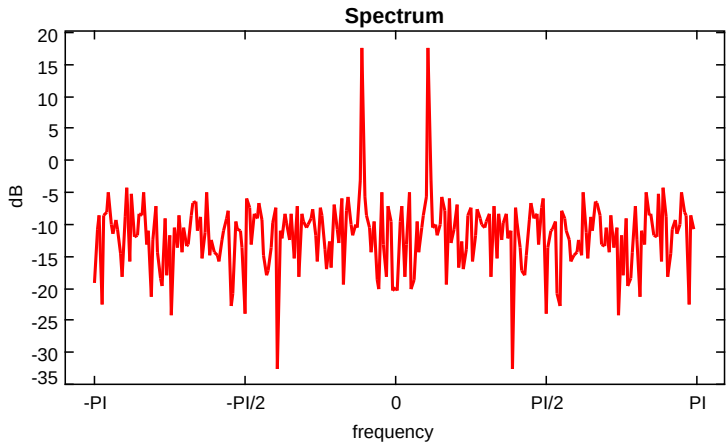


Figure 17.12: Still better labeled plot.

Bibliography

Agha, G. A., I. A. Mason, S. F. Smith, and C. L. Talcott, 1997: A foundation for actor computation. *Journal of Functional Programming*, **7(1)**, 1–72.

Allen, F. E., 1970: Control flow analysis. *SIGPLAN Notices*, **5(7)**, 1–19.

Alur, R., S. Kannan, and M. Yannakakis, 1999: Communicating hierarchical state machines. In *26th International Colloquium on Automata, Languages, and Programming*, Springer, vol. LNCS 1644, pp. 169–178.

Andalam, S., P. S. Roop, and A. Girault, 2010: Predictable multithreading of embedded applications using PRET-C. In *Formal Methods and Models for Codesign (MEMOCODE)*, IEEE/ACM, Grenoble, France, pp. 159–168. doi:10.1109/MEMCOD.2010.5558636.

André, C., 1996: SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, revision: RR (96–56), University of Sophia-Antipolis. Available from: http://www-sop.inria.fr/members/Charles.Andre/CA%20Publis/SYNCCHARTS/overview.html.

André, C., F. Mallet, and R. d. Simone, 2007: Modeling time(s). In *Model Driven Engineering Languages and Systems (MoDELS/UML)*, Springer, Nashville, TN, vol. LNCS 4735, pp. 559–573. doi:10.1007/978-3-540-75209-7_38.

Arbab, F., 2006: A behavioral model for composition of software components. *L'Object, Lavoisier*, **12(1)**, 33–76. doi:10.3166/objet.12.1.33-76.

Arvind, L. Bic, and T. Ungerer, 1991: Evolution of data-flow computers. In Gaudiot, J.-L. and L. Bic, eds., *Advanced Topics in Data-Flow Computing*, Prentice-Hall.

Baccelli, F., G. Cohen, G. J. Olster, and J. P. Quadrat, 1992: *Synchronization and Linearity, An Algebra for Discrete Event Systems*. Wiley, New York.

Baier, C. and M. E. Majster-Cederbaum, 1994: Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, **135(2)**, 171–220.

Balarin, F., H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, 2003: Metropolis: an integrated electronic system design environment. *Computer*, **36(4)**.

Baldwin, P., S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, 2004: Modeling of sensor nets in Ptolemy II. In *Information Processing in Sensor Networks (IPSN)*, Berkeley, CA, USA. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/04/VisualSense/.

—, 2005: Visualsense: Visual modeling for wireless and sensor network systems. Technical Report UCB/ERL M05/25, EECS Department, University of California. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/05/visualsense/index.htm.

Basu, A., M. Bozga, and J. Sifakis, 2006: Modeling heterogeneous real-time components in BIP. In *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, pp. 3–12.

Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.

Benveniste, A., P. Caspi, P. Le Guernic, and N. Halbwachs, 1994: Data-flow synchronous languages. In Bakker, J. W. d., W.-P. d. Roever, and G. Rozenberg, eds., *A Decade of Concurrency Reflections and Perspectives*, Springer-Verlag, Berlin, vol. 803 of *LNCS*, pp. 1–45.

Benveniste, A. and P. Le Guernic, 1990: Hybrid dynamical systems theory and the SIGNAL language. *IEEE Tr. on Automatic Control*, **35(5)**, 525–546.

Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Franaise dAutomatique, Informatique et Recherche Oprationnelle*, **10(3)**, 47–82.

—, 1999: *The Constructive Semantics of Pure Esterel - Draft Version 3*. Book Draft. Available from: http://www-sop.inria.fr/meije/esterel/doc/main-papers.html.

—, 2003: The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies. Available from: http://www.esterel-technologies.com.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, **19(2)**, 87–152. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.5606.

Bhattacharya, B. and S. S. Bhattacharyya, 2000: Parameterized dataflow modeling of DSP systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Istanbul, Turkey, pp. 1948–1951.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. Lee, 1995: Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **42(3)**, 138–150. doi:10.1109/81.376876.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. A. Lee, 1993: A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *VLSI Signal Processing VI*, IEEE, Veldhoven , The Netherlands, pp. 188–196. doi:10.1109/VLSISP.1993.404488.

Bhattacharyya, S. S. and E. A. Lee, 1993: Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, **6(3)**, 271–288. doi:10.1007/BF01608539.

Bhattacharyya, S. S., P. Murthy, and E. A. Lee, 1996a: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, **2(1)**, 33–60. doi:10.1023/A:1008806425898.

Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee, 1996b: *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, Mass.

Bilsen, G., M. Engels, R. Lauwereins, and J. A. Peperstraete, 1996: Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, **44(2)**, 397–408. doi:10.1109/78.485935.

Bock, C., 2006: SysML and UML 2 support for activity modeling. *Systems Engineering*, **9(2)**, 160 –185.

Booch, G., I. Jacobson, and J. Rumbaugh, 1998: *The Unified Modeling Language User Guide*. Addison-Wesley.

Boussinot, F., 1991: Reactive c: An extension to c to program reactive systems. *Software Practice and Experience*, **21(4)**, 401–428.

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley.

Brock, J. D. and W. B. Ackerman, 1981: Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, Springer-Verlag, vol. LNCS 107, pp. 252–259.

Broenink, J. F., 1997: Modelling, simulation and analysis with 20-Sim. *CACSD*, **38(3)**, 22–25.

Brooks, C., C. Cheng, T. H. Feng, E. A. Lee, and R. von Hanxleden, 2008: Model engineering using multimodeling. In *International Workshop on Model Co-Evolution and Consistency Management (MCCM)*, Toulouse, France. Available from: http://chess.eecs.berkeley.edu/pubs/486.html.

Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, 2004: Heterogeneous concurrent modeling and design in Java. Tech. Rep. Technical Memorandum UCB/ERL M04/16, University of California. Available from: http://ptolemy.eecs.berkeley.edu/papers/04/ptIIDesignSoftware/.

Brooks, C. H. and E. A. Lee, 2003: Ptolemy II coding style. Tech. Rep. Technical Memorandum UCB/ERL M03/44, University of California at Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/03/codingstyle/.

Broy, M., 1983: Applicative real time programming. In *Information Processing 83, IFIP World Congress*, North Holland Publ. Company, Paris, pp. 259–264.

Broy, M. and G. Stefanescu, 2001: The algebra of stream processing functions. *Theoretical Computer Science*, **258**, 99–129.

Bryant, V., 1985: *Metric Spaces - Iteration and Application*. Cambridge University Press.

Buck, J. T., 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Tech. Report UCB/ERL 93/69, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/.

Buck, J. T., S. Ha, E. A. Lee, and D. G. Messerschmitt, 1994: Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, **4**, 155–182. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/94/JEurSim/.

Burch, J. R., R. Passerone, and A. L. Sangiovanni-Vincentelli, 2001: Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *International Conference on Application of Concurrency to System Design*, p. 13.

Buss, A. H. and P. J. Sanchez, 2002: Building complex models with LEGOs (listener event graph objects). *Winter Simulation Conference (WSC 02)*, **1**, 732–737.

Cardelli, L., 1997: Type systems. In Tucker, A. B., ed., *The Computer Science and Engineering Handbook*, CRC Press, chap. 103, pp. 2208–2236, http://lucacardelli.name/Papers/TypeSystems

Cardelli, L. and P. Wegner, 1985: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, **17(4)**, 471 – 523.

Carloni, L. P., R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, 2006: Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, **1(1/2)**. doi:10.1561/1000000001.

Caspi, P., P. Raymond, and S. Tripakis, 2007: Synchronous Programming. In Lee, I., J. Leung, and S. Son, eds., *Handbook of Real-Time and Embedded Systems*, Chapman & Hall, pp. 14–1 — 14–21. Available from: http://www-verimag.imag.fr/~tripakis/papers/handbook07.pdf.

Cassandras, C. G., 1993: *Discrete Event Systems, Modeling and Performance Analysis*. Irwin.

Cataldo, A., E. A. Lee, X. Liu, E. Matsikoudis, and H. Zheng, 2006: A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/06/constructive/.

Chandy, K. M. and J. Misra, 1979: Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, **5(5)**, 440–452.

Clarke, E. M., O. Grumberg, and D. A. Peled, 2000: *Model checking*. MIT Press, ISBN 0-262-03270-8.

Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.

Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6(7)**, 396–408.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, 2012: Spanner: Googles globally-distributed database. In *OSDI*.

Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.

Creeger, M., 2005: Multicore CPUs for the masses. *ACM Queue*, **3(7)**, 63–64.

Davey, B. A. and H. A. Priestly, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.

de Alfaro, L. and T. Henzinger, 2001: Interface automata. In *ESEC/FSE 01: the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

Dennis, J. B., 1974: First version data flow procedure language. Tech. Rep. MAC TM61, MIT Laboratory for Computer Science.

Derler, P., E. A. Lee, and S. Matic, 2008: Simulation and implementation of the ptides programming model. In *IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Vancouver, Canada.

Dijkstra, E. W., 1968: Go to statement considered harmful (letter to the editor). *Communications of the ACM*, **11(3)**, 147–148.

Edwards, S. A. and E. A. Lee, 2003a: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

—, 2003b: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/blockdiagram/.

Eidson, J. C., 2006: *Measurement, Control, and Communication Using IEEE 1588*. Springer. doi:10.1007/1-84628-251-9.

Eidson, J. C., E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, 2012: Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, **100(1)**, 45–59. doi:10.1109/JPROC.2011.2161237.

Eker, J. and J. W. Janneck, 2003: Cal language report: Specification of the cal actor language. Tech. Rep. Technical Memorandum No. UCB/ERL M03/48, University of California, Berkeley, CA. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/Cal/index.htm.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, **91(2)**, 127–144. Available from: http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/.

Encyclopedia Britannica, 2010: Ockham's razor. *Encyclopedia Britannica Online, Retrieved June 24, 2010*. Available from: http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor.

Falk, J., J. Keiner, C. Haubelt, J. Teich, and S. S. Bhattacharyya, 2008: A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Embedded Software (EMSOFT)*, ACM, Atlanta, Georgia, USA.

Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.

Feng, T. H., 2009: Model transformation with hierarchical discrete-event control. PhD Thesis UCB/EECS-2009-77, EECS Department, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-77.html.

Feng, T. H. and E. A. Lee, 2008: Real-time distributed discrete-event execution with fault tolerance. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, St. Louis, MO, USA. Available from: http://chess.eecs.berkeley.edu/pubs/389.html.

Feng, T. H., E. A. Lee, H. D. Patel, and J. Zou, 2008: Toward an effective execution policy for distributed real-time embedded systems. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, St. Louis, MO, USA. Available from: https://chess.eecs.berkeley.edu/pubs/402.

Feng, T. H., E. A. Lee, and L. W. Schruben, 2010: Ptera: An event-oriented model of computation for heterogeneous systems. In *EMSOFT*, ACM Press, Scottsdale, Arizona, USA. doi:10.1145/1879021.1879050.

Feredj, M., F. Boulanger, and A. M. Mbobi, 2009: A model of domain-polymorph component for heterogeneous system design. *The Journal of Systems and Software*, **82**, 112–120.

Fitzgerald, J., P. G. Larsen, K. Pierce, M. Verhoef, and S. Wolff, 2010: Collaborative modelling and co-simulation in the development of dependable embedded systems. In *Integrated Formal Methods (IFM)*, Springer-Verlag, vol. LNCS 6396, pp. 12–26. doi:10.1007/978-3-642-16265-7_2.

Fitzgerald, J. S., P. G. Larsen, and M. Verhoef, 2008: Vienna development method. In *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc. doi:10.1002/9780470050118.ecse447.

Foley, J., A. van Dam, S. Feiner, and J. Hughes, 1996: *Computer Graphics, Principles and Practice*. Addisson-Wesley, 2nd ed.

Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.

Fritzson, P., 2003: *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley.

Fuhrmann, H. and R. v. Hanxleden, 2010: Taming graphical modeling. In *Model Driven Engineering Languages and Systems (MODELS) 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010*, Springer, Oslo, Norway, vol. 6394, pp. 196–210. doi:10.1007/978-3-642-16145-2_14.

Fuhrmann, H. and R. von Hanxleden, 2008: On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development – Monterey Workshop*, Springer, Budapest, Hungary, vol. LNCS 6028, pp. 116–140. doi:10.1007/978-3-642-12566-9_7.

Fujimoto, R., 2000: *Parallel and Distributed Simulation Systems*. John Wiley and Sons.

Gaderer, G., P. Loschmidt, E. G. Cota, J. H. Lewis, J. Serrano, M. Cattin, P. Alvarez, P. M. Oliveira Fernandes Moreira, T. Wlostowski, J. Dedic, C. Prados, M. Kreider, R.Baer, S.Rauch, and T.Fleck, 2009: The white rabbit project. In *Int. Conf. on Accelerator and Large Experimental Physics Control Systems*, Kobe, Japan.

Galletly, J., 1996: *Occam-2*. University College London Press, 2nd ed.

Ganter, B. and R. Wille, 1998: *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334. Available from: http://www.ics.ele.tue.nl/~tbasten/papers/esop03.pdf.

Geilen, M., T. Basten, and S. Stuijk, 2005: Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 819–824. doi:10.1145/1065579.1065796.

Geilen, M. and S. Stuijk, 2010: Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, ACM, Scottsdale, Arizona, USA, pp. 125–134.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.

Goderis, A., C. Brooks, I. Altintas, E. A. Lee, and C. Goble, 2009: Heterogeneous composition of models of computation. *Future Generation Computer Systems*, **25(5)**, 552–560. doi:doi:10.1016/j.future.2008.06.014.

Goessler, G. and A. Sangiovanni-Vincentelli, 2002: Compositional modeling in Metropolis. In *Second International Workshop on Embedded Software (EMSOFT)*, Springer-Verlag, Grenoble, France.

Golomb, S. W., 1971: Mathematical models: Uses and limitations. *IEEE Transactions on Reliability*, **R-20(3)**, 130–131. doi:10.1109/TR.1971.5216113.

Gu, Z., S. Wang, S. Kodase, and K. G. Shin, 2003: An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In *Real-Time Systems Symposium (RTSS)*, pp. 78 – 81.

Ha, S. and E. A. Lee, 1991: Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, **40(11)**, 1225–1238. doi:10.1109/12.102826.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud, 1991: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, **79(9)**, 1305–1319.

Hardebolle, C. and F. Boulanger, 2007: ModHel'X: A component-oriented approach to multi- formalism modeling. In *MODELS 2007 Workshop on Multi- Paradigm Modeling*, Elsevier Science B.V., Nashville, Tennessee, USA.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8(3)**, 231–274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, **16(4)**, 403 – 414. doi:10.1109/32.54292.

Harel, D. and A. Pnueli, 1985: On the development of reactive systems. In Apt, K. R., ed., *Logic and Models for Verification and Specification of Concurrent Systems*, Springer-Verlag, vol. F13 of *NATO ASI Series*, pp. 477–498.

Henzinger, T. A., 2000: The theory of hybrid automata. In Inan, M. and R. Kurshan, eds., *Verification of Digital and Hybrid Systems*, Springer-Verlag, vol. 170 of *NATO ASI Series F: Computer and Systems Sciences*, pp. 265–292.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch, 2001: Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Springer-Verlag, Tahoe City, CA, vol. LNCS 2211, pp. 166–184.

Herrera, F. and E. Villar, 2006: A framework for embedded system specification under different models of computation in SystemC. In *Design Automation Conference (DAC)*, ACM, San Francisco.

Hewitt, C., 1977: Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, **8(3)**, 323–363.

Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Hsu, C.-J., F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, 2004: DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece.

Hu, T. C., 1961: Parallel sequencing and assembly line problems. *Operations Research*, **9(6)**, 841–848.

Ingalls, R. G., D. J. Morrice, and A. B. Whinston, 1996: Eliminating canceling edges from the simulation graph model methodology. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, IEEE Computer Society, Washington, DC, USA, ISBN 0-7803-3383-7, pp. 825–832.

Jantsch, A., 2003: *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

Jantsch, A. and I. Sander, 2005: Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, **152(2)**, 114–129.

Jefferson, D., 1985: Virtual time. *ACM Trans. Programming Languages and Systems*, **7(3)**, 404–425.

Johannessen, S., 2004: Time synchronization in a local area network. *IEEE Control Systems Magazine*, 61–69.

Johnston, W. M., J. R. P. Hanna, and R. J. Millar, 2004: Advances in dataflow programming languages. *ACM Computing Surveys*, **36(1)**, 1–34.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., pp. 993–998.

Karsai, G., A. Lang, and S. Neema, 2005: Design patterns for open tool integration. *Software and Systems Modeling*, **4(2)**, 157–170. doi:10.1007/s10270-004-0073-y.

Kay, S. M., 1988: *Modern Spectral Estimation: Theory & Application*. Prentice-Hall, Englewood Cliffs, NJ.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, 1997: Aspect-oriented programming. In *ECOOP, European Conference in Object-Oriented Programming*, Springer-Verlag, Finland, vol. LNCS 1241.

Kienhuis, B., E. Deprettere, P. van der Wolf, and K. Vissers, 2001: A methodology to design programmable embedded systems. In Deprettere, E., J. Teich, and S. Vassiliadis, eds., *Systems, Architectures, Modeling, and Simulation (SAMOS)*, Springer-Verlag, vol. LNCS 2268.

Kodosky, J., J. MacCrisken, and G. Rymar, 1991: Visual programming using structured data flow. In *IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 34–39.

Kopetz, H., 1997: *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.

Kopetz, H. and G. Bauer, 2003: The time-triggered architecture. *Proceedings of the IEEE*, **91(1)**, 112–126.

Lamport, L., R. Shostak, and M. Pease, 1978: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21(7)**, 558–565.

Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.

Le Guernic, P., T. Gauthier, M. Le Borgne, and C. Le Maire, 1991: Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, **79(9)**, 1321 – 1336. doi:10.1109/5.97301.

Lee, E. A., 1986: A coupled hardware and software architecture for programmable digital signal processors. PhD Thesis UCB/ERL M86/54, University of California. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/86/LeePhDThesis/.

—, 1999: Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45. doi:10.1023/A:1018998524196.

—, 2006: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.

—, 2008a: Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, Orlando, Florida, pp. 363 – 369. doi:10.1109/ISORC.2008.25.

—, 2008b: ThreadedComposite: A mechanism for building concurrent and parallel Ptolemy II models. Technical Report UCB/EECS-2008-151, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-151.html.

—, 2009: Finite state machines and modal models in Ptolemy II. Tech. Rep. UCB/EECS-2009-151, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html.

—, 2010a: CPS foundations. In *Design Automation Conference (DAC)*, ACM, Anaheim, California, USA, pp. 737–742. doi:10.1145/1837274.1837462.

—, 2010b: Disciplined heterogeneous modeling. In Petriu, D. C., N. Rouquette, and O. Haugen, eds., *Model Driven Engineering, Languages, and Systems (MODELS)*, IEEE, pp. 273–287. Available from: http://chess.eecs.berkeley.edu/pubs/679.html.

Lee, E. A., E. Goei, H. Heine, and W. Ho, 1989: Gabriel: A design environment for programmable DSPs. In *Design Automation Conference (DAC)*, Las Vegas, NV, pp. 141–146. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/89/gabriel/.

Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279 –1283. doi:10.1109/GLOCOM.1989.64160.

Lee, E. A., X. Liu, and S. Neuendorffer, 2009a: Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, **8(4)**, 29:1–29:26. doi:10.1145/1550987.1550992.

Lee, E. A., S. Matic, S. A. Seshia, and J. Zou, 2009b: The case for timing-centric distributed software. In *IEEE International Conference on Distributed Computing Systems Workshops: Workshop on Cyber-Physical Systems*, IEEE, Montreal, Canada, pp. 57–64. Available from: http://chess.eecs.berkeley.edu/pubs/607.html.

Lee, E. A. and E. Matsikoudis, 2009: The semantics of dataflow with firing. In Huet, G., G. Plotkin, J.-J. Lévy, and Y. Bertot, eds., *From Semantics to Computer Science: Essays in memory of Gilles Kahn*, Cambridge University Press. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

Lee, E. A. and D. G. Messerschmitt, 1987a: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **C-36(1)**, 24–35. doi:10.1109/TC.1987.5009446.

—, 1987b: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.

Lee, E. A. and S. Neuendorffer, 2000: MoML - a modeling markup language in XML. Tech. Rep. UCB/ERL M00/12, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/.

Lee, E. A., S. Neuendorffer, and M. J. Wirthlin, 2003: Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, **12(3)**, 231–260. Available from: http://ptolemy.eecs.berkeley.edu/papers/03/actorOrientedDesign/.

Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.

Lee, E. A. and A. Sangiovanni-Vincentelli, 1998: A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*,

17(12), 1217–1229. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/.

Lee, E. A. and S. A. Seshia, 2011: *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. LeeSeshia.org, Berkeley, CA. Available from: http://LeeSeshia.org.

Lee, E. A. and S. Tripakis, 2010: Modal models in Ptolemy. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, Linköping University Electronic Press, Linköping University, Oslo, Norway, vol. 47, pp. 11–21. Available from: http://chess.eecs.berkeley.edu/pubs/700.html.

Lee, E. A. and P. Varaiya, 2011: *Structure and Interpretation of Signals and Systems*. LeeVaraiya.org, 2nd ed. Available from: http://LeeVaraiya.org.

Lee, E. A. and H. Zheng, 2005: Operational semantics of hybrid systems. In Morari, M. and L. Thiele, eds., *Hybrid Systems: Computation and Control (HSCC)*, Springer-Verlag, Zurich, Switzerland, vol. LNCS 3414, pp. 25–53. doi:10.1007/978-3-540-31954-2_2.

—, 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, Salzburg, Austria, pp. 114 – 123. doi:10.1145/1289927.1289949.

Leung, M.-K., T. Mandl, E. A. Lee, E. Latronico, C. Shelton, S. Tripakis, and B. Lickly, 2009: Scalable semantic annotation using lattice-based ontologies. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, ACM/IEEE, Denver, CO, USA. Available from: http://chess.eecs.berkeley.edu/pubs/611.html.

Lickly, B., 2012: Static model analysis with lattice-based ontologies. PhD Thesis Technical Report No. UCB/EECS-2012-212, EECS Department, University of California, Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-212.html.

Lickly, B., C. Shelton, E. Latronico, and E. A. Lee, 2011: A practical ontology framework for static model analysis. In *International Conference on Embedded Software (EMSOFT)*, ACM, pp. 23–32. Available from: http://chess.eecs.berkeley.edu/pubs/862.html.

Lin, Y., R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, 2006: SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, Orlando. Available from: http://www.eecs.umich.edu/~sdrg/publications.php.

Liskov, B. and S. Zilles, 1974: Programming with abstract data types. *ACM Sigplan Notices*, **9(4)**, 50–59. doi:10.1145/942572.807045.

Liu, J., B. Wu, X. Liu, and E. A. Lee, 1999: Interoperation of heterogeneous CAD tools in Ptolemy II. In *Symposium on Design, Test, and Microfabrication of MEMS/MOEMS*, Paris, France. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/99/toolinteraction/.

Liu, X. and E. A. Lee, 2008: CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, **409(1)**, 110–125. doi:10.1016/j.tcs.2008.08.044.

Liu, X., E. Matsikoudis, and E. A. Lee, 2006: Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, Springer, Bonn, Germany, vol. LNCS 4137, pp. 1–15. doi:10.1007/11817949_1.

Lynch, N., R. Segala, F. Vaandrager, and H. Weinberg, 1996: Hybrid I/O automata. In Alur, R., T. Henzinger, and E. Sontag, eds., *Hybrid Systems III*, Springer-Verlag, vol. LNCS 1066, pp. 496–510.

Lzaro Cuadrado, D., A. P. Ravn, and P. Koch, 2007: Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*, Acta Press.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, Springer-Verlag, pp. 447–484.

Malik, S., 1994: Analysis of cyclic combinational circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, **13(7)**, 950–956.

Manna, Z. and A. Pnueli, 1992: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin.

—, 1993: Verifying hybrid systems. In *Hybrid Systems*, vol. LNCS 736, pp. 4–35.

Maraninchi, F. and T. Bhouhadiba, 2007: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *6th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, Salzburg, Austria, pp. 1–3.

Maraninchi, F. and Y. Rémond, 2001: Argos: an automaton-based synchronous language. *Computer Languages*, **(27)**, 61–92.

Matic, S., I. Akkaya, M. Zimmer, J. C. Eidson, and E. A. Lee, 2011: Ptides model on a distributed testbed emulating smart grid real-time applications. In *Innovative Smart Grid Technologies (ISGT-EUROPE)*, IEEE, Manchester, UK. Available from: http://chess.eecs.berkeley.edu/pubs/857.html.

Matsikoudis, E., C. Stergiou, and E. A. Lee, 2013: On the schedulability of real-time discrete-event systems. In *International Conference on Embedded Software (EMSOFT)*, ACM, Montreal, Canada.

Matthews, S. G., 1995: An extensional treatment of lazy data flow deadlock. *Theoretical Computer Science*, **151(1)**, 195–205.

Messerschmitt, D. G., 1984: A tool for structured functional simulation. *IEEE Journal on Selected Areas in Communications*, **SAC-2(1)**.

Mills, D. L., 2003: A brief history of NTP time: confessions of an internet timekeeper. *ACM Computer Communications Review*, **33**.

Milner, R., 1978: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**, 348–375.

—, 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer.

Misra, J., 1986: Distributed discrete event simulation. *ACM Computing Surveys*, **18(1)**, 39–65.

Modelica Association, 2009: Modelica®- a unified object-oriented language for physical systems modeling: Language specification version 3.1. Report. Available from: http://www.Modelica.org.

Moir, I. and A. Seabridge, 2008: *Aircraft Systems: Mechanical, Electrical, and Avionics Subsystems Integration*. AIAA Education Series, Wiley, third edition ed.

Moreira, O., T. Basten, M. Geilen, and S. Stuijk, 2010: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, **59(2)**, 188–201. doi:10.1109/TC.2009.155.

Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.

Mosterman, P. J. and H. Vangheluwe, 2004: Computer automated multi-paradigm modeling: An introduction. *Simulation: Transactions of the Society for Modeling and Simulation International Journal of High Performance Computing Applications*, **80(9)**, 433–450.

Motika, C., H. Fuhrmann, and R. v. Hanxleden, 2010: Semantics and execution of domain specific models. In *Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) at conference INFORMATIK 2010*, Bonner Köllen Verlag, Leipzig, Germany, vol. GI-Edition – Lecture Notes in Informatics (LNI),.

Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77(4)**, 541–580. doi:10.1109/5.24143.

Murthy, P. K. and S. S. Bhattacharyya, 2006: *Memory Management for Synthesis of DSP Software*. CRC Press.

Murthy, P. K. and E. A. Lee, 2002: Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, **50(8)**, 2064–2079. doi:10.1109/TSP.2002.800830.

Object Management Group (OMG), 2007: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG. Available from: http://www.omg.org/omgmarte/.

—, 2008a: System modeling language specification v1.1. Tech. rep., OMG. Available from: http://www.sysmlforum.com.

—, 2008b: A UML profile for MARTE, beta 2. OMG Adopted Specification ptc/08-06-09, OMG. Available from: http://www.omg.org/omgmarte/.

Olson, A. G. and B. L. Evans, 2005: Deadlock detection for distributed process networks. In *ICASSP*.

Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis.

Parks, T. M. and D. Roberts, 2003: Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, Nice, France.

Patel, H. D. and S. K. Shukla, 2004: *SystemC Kernel Extensions for Heterogeneous System Modelling*. Kluwer.

Pino, J. L., T. M. Parks, and E. A. Lee, 1994: Automatic code generation for heterogeneous multiprocessors. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Adelaide, Australia, pp. 445–448. doi:10.1109/ICASSP.1994.389626.

Pree, W. and J. Templ, 2006: Modeling with the timing definition language (TDL). In *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, Springer, San Diego, CA, LNCS.

Press, W. H., S. Teukolsky, W. T. Vetterling, and B. P. Flannery, 1992: *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press.

Prochnow, S. and R. von Hanxleden, 2007: Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, ACM/IEEE, Nashville, TN, USA.

Ramadge, P. and W. Wonham, 1989: The control of discrete event systems. *Proceedings of the IEEE*, **77(1)**, 81–98.

Reed, G. M. and A. W. Roscoe, 1988: Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, London, UK, pp. 331–343.

Rehof, J. and T. A. Mogensen, 1996: Tractable constraints in finite semilattices. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, Springer-Verlag, London, UK, ISBN 3-540-61739-6, pp. 285–300.

Rehof, J. and T. . Mogensen, 1999: Tractable constraints in finite semilattices. *Science of Computer Programming*, **35(2-3)**, 191–221.

Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17(7)**, 365 – 375.

Rodiers, B. and B. Lickly, 2010: Width inference documentation. Technical Report UCB/EECS-2010-120, EECS Department, University of California. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-120.html.

Sander, I. and A. Jantsch, 2004: System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **23(1)**, 17–32.

Schruben, L. W., 1983: Simulation modeling with event graphs. *Communications of the ACM*, **26(11)**, 957–963.

—, 1995: Building reusable simulators using hierarchical event graphs. In *Winter Simulation Conference (WSC 95)*, IEEE Computer Society, Los Alamitos, CA, USA, ISBN 0-7803-3018-8, pp. 472–475.

Shapiro, F. R., 2006: *The Yale Book of Quotations*. Yale University Press.

Sih, G. C. and E. A. Lee, 1993a: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, **4(2)**, 175–187. doi:10.1109/71.207593.

—, 1993b: Declustering : A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, **4(6)**, 625–637. doi:10.1109/71.242160.

Simitci, H., 2003: *Storage Network Performance Analytics*. Wiley.

Smith, N. K., 1929: *Immanuel Kant's Critique of Pure Reason*. Macmillan and Co. Available from: http://www.hkbu.edu.hk/~ppp/cpr/toc.html.

Som, T. K. and R. G. Sargent, 1989: A formal development of event graph models as an aid to structured and efficient simulation programs. *ORSA Journal on Computing*, **1(2)**, 107–125.

Spönemann, M., H. Fuhrmann, R. v. Hanxleden, and P. Mutzel, 2009: Port constraints in hierarchical layout of data flow diagrams. In *17th International Symposium on Graph Drawing (GD)*, Springer, Chicago, IL, USA, vol. LNCS. Available from: http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/gd09.pdf.

Srini, V., 1986: An architectural comparison of dataflow systems. *Computer*, **19(3)**.

Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22(1-2)**, 167–185.

Stephens, R., 1997: A survey of stream processing. *Acta Informatica*, **34(7)**.

Stuijk, S., M. C. Geilen, and T. Basten, 2008: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, **57(10)**, 1331–1345. doi:10.1109/TC.2008.58.

Thies, W., M. Karczmarek, and S. Amarasinghe, 2002: StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, Springer-Verlag, Grenoble, France, vol. LNCS 2304. doi:10.1007/3-540-45937-5_14.

Thies, W., M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, 2005: Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming (PPoPP)*, ACM, Chicago, USA. doi:10.1145/1065944.1065975.

Tripakis, S., C. Stergiou, C. Shaver, and E. A. Lee, 2013: A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, **23**, 834–881. Available from: http://chess.eecs.berkeley.edu/pubs/999.html, doi:10.1017/S0960129512000278.

Turjan, A., B. Kienhuis, and E. Deprettere, 2003: Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, **40**, 7 – 18. doi:10.1007/s11265-005-4935-5.

University of Pennsylvania MoBIES team, 2002: HSIF semantics (version 3, synchronous edition). Tech. Rep. Report, University of Pennsylvania.

von der Beeck, M., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytopil, eds., *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer-Verlag, Lübeck, Germany, vol. 863 of *Lecture Notes in Computer Science*, pp. 128–148.

von Hanxleden, R., 2009: SyncCharts in C - A proposal for light-weight deterministic concurrency. In *ACM Embedded Software Conference (EMSOFT)*, pp. 11–16. doi:10.1145/1629335.1629366.

Wiener, N., 1948: *Cybernetics: Or Control and Communication in the Animal and the Machine*. Librairie Hermann & Cie, Paris, and MIT Press.Cambridge, MA.

Xiong, Y., 2002: An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720. Available from: http://ptolemy.eecs.berkeley.edu/papers/02/typeSystem.

Yates, R. K., 1993: Networks of real-time processes. In Best, E., ed., *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, Springer-Verlag, vol. LNCS 715.

Zeigler, B., 1976: *Theory of Modeling and Simulation*. Wiley Interscience, New York.

Zeigler, B. P., H. Praehofer, and T. G. Kim, 2000: *Theory of Modeling and Simulation*. Academic Press, 2nd ed.

Zhao, Y., 2009: On the design of concurrent, distributed real-time systems. Ph.D. Thesis Technical Report UCB/EECS-2009-117, EECS Department, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-117.html.

Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, Bellevue, WA, USA, pp. 259 – 268. doi:10.1109/RTAS.2007.5.

Zou, J., 2011: From ptides to ptidyos, designing distributed real-time embedded systems. PhD Dissertation Technical Report UCB/EECS-2011-53, UC Berkeley. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-53.html.

Zou, J., J. Auerbach, D. F. Bacon, and E. A. Lee, 2009a: PTIDES on flexible task graph: Real-time embedded system building from theory to practice. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM, Dublin, Ireland. Available from: http://chess.eecs.berkeley.edu/pubs/531.html.

Zou, J., S. Matic, E. A. Lee, T. H. Feng, and P. Derler, 2009b: Execution strategies for Ptides, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, San Francisco, CA. Available from: http://chess.eecs.berkeley.edu/pubs/529.html.

Actor Index

A small fraction of the actors in the standard Ptolemy II library are described in detail in this book. Below is a summary of all the actors in the library.

Sources Library		
Const	produce a constant sequence	48
CurrentMicrostep	produce the current microstep when fired	242
CurrentTime	produce the current model time when fired	242
DiscreteClock	produce a timed sequence	241
InteractiveShell	shell for interaction with a user	136
Interpolator	produce a signal by interpolating given values	48
Pulse	produce a sequence pattern	48
Ramp	produce a counting sequence	48
Sequence	produce a sequence of specified values	48
SketchedSource	produce a sketched signal	48
StringConst	produce a constant string-valued sequence	48
Subscriber	output tokens published by a Publisher	48
SubscriptionAggregator	output tokens published by a Publisher	48
PoissonClock	produce a random timed sequence	241
TriggeredSinewave	produce samples of a sine wave	48
Sinewave	produce a sine wave	48

Sinks Library		
ArrayPlotter	plot an input array	601
ArrayPlotterXY	plot an array input as an X-Y plot	601
BarGraph	create a bar graph	601
Discard	discard the input	49
Display	display the input value in a window	49
HistogramPlotter	create a histogram	601
MonitorValue	display the input value in the icon	248
Publisher	send inputs to Subscribers	49
Recorder	record inputs	49
SequencePlotter	Plot an input sequence	600
SequenceScope	Plot an input sequence	601
SetVariable	set a variable equal the value of the input	49
TimedPlotter	plot an input sequence vs. time	601
TimedScope	plot an input sequence vs. time	601
XYPlotter	plot an input sequence as an X-Y plot	601
XYScope	plot an input sequence as an X-Y plot	601

Array Library		
ArrayAccumulate	accumulate input arrays into one array	88
ArrayAppend	append input arrays into one array	88
ArrayAverage	average the elements of an array	88
ArrayContains	determine whether an array contains an element	88
ArrayElement	extract an element of an array	88
ArrayElementAsMatrix	extract an element of an array	88
ArrayExtract	extract a subarray	88
ArrayLength	output the length of an array	88
ArrayLevelCrossing	search a subarray for a level crossing	88
ArrayMaximum	find the maximum value in an array	88
ArrayMinimum	find the minimum value in an array	88
ArrayPeakSearch	search an array for peaks	88
ArrayRemoveElement	remove the specified element from an array	88
ArraySort	sort an array	88
ArraySum	sum elements of an array	88
ArrayToElements	break an array into elements	86
ArrayToSequence	output the array elements in sequence	86
ArrayUpdate	change an element of an array	88
ElementsToArray	construct an array from elements	86
SequenceToArray	convert a sequence into an array	86

Conversions		
AnythingToDouble	cast to double (with NaN for non-doubles)	—
BooleanToAnything	convert a boolean to two arbitrary values	169
BitsToInt	convert a sequence of bits to an integer	—
CartesianToComplex	convert two doubles to complex	—
CartesianToPolar	convert cartesian to magnitude and phase	—
ComplexToCartesian	convert complex to two doubles	—
ComplexToPolar	convert complex to magnitude and phase	—
DoubleToFix	convert a double to a fixed-point number	—
ExpressionToToken	parse and evaluate a string expression	253
FixToDouble	convert a fixed-point number to a double	—
FixToFix	convert a fixed-point number to another	—
IntToBits	convert an integer to a sequence of bits	—
InUnitsOf	reinterpret a value in new units	—
LongToDouble	coerce a long into a double	—
PolarToCartesian	convert magnitude and phase to cartesian	—
Round	convert a double to an int	—
StringToUnsignedByteArray	convert a string to an array of bytes	86
StringToXML	parse an XML-formatted string	—
TokenToExpression	produce a string representation of a token	634
UnsignedByteArrayToString	convert a byte array to a string	86

Flow Control		
BooleanMultiplexor	multiplexor with two inputs	119
BooleanSelect	Select with two inputs	119
BooleanSwitch	Switch with two outputs	119
BusAssembler	aggregate signals into a bus	65
BusDisassembler	disaggregate signals from a bus	65
Chop	break a sequence into chunks	106
Commutator	interleave streams in round robin	106
ConfigurationSelect	select inputs based on a parameter	119
ConfigurationSwitch	switch signals based on a parameter	119
CountTrues	output the number of trues received	—
Distributor	divide a stream in round robin	106
Exit	stop the current Ptolemy process and exit	—
Multiplexor	build a stream with elements from streams	119
OrderedRecordAssembler	construct an ordered record	253
RecordAssembler	construct a record	253
RecordDisassembler	extract fields from a record	253
RecordUpdater	update fields in a record	253

Flow Control		
Repeat	repeat an input token	106
Sampler	sample a signal	244
SampleDelay	output initial tokens	103
Select	interleave streams	119
Sequencer	sequence tokens by sequence number	143
SingleTokenCommutator	commutator that outputs one token at a time	110
Stop	stop a model execution	143
Switch	split streams	119
Synchronizer	synchronize a set of streams	143
ThrowException	throw an exception	—
ThrowModelError	throw a model error	—
UnionDisassembler	extract a particular type from a union	471
UnionMerge	merge types into a union	471
VectorAssembler	construct a column or row matrix	—
VectorDisassembler	deconstruct a column or row matrix	—

Higher Order Actors		
ApplyFunction	apply a function to the inputs	89
Case	apply one of n models to the inputs	120
IterateOverArray	apply a model to each element of an input array	84
MobileModel	apply a model provided as an input	89
ModelDisplay	display a model	—
ModelReference	execute a model defined in another file	87
MultiInstanceComposite	execute copies of a model on inputs	81
RunCompositeActor	execute a submodel to completion	87
PtalonActor	construct a model using Ptalon	—
ThreadedComposite	execute a model in a new thread	258
VisualModelReference	execute a model defined in another file	87

IO Actors		
ArrowKeySensor	report keystrokes on the arrow keys	128
CSVReader	read comma-separated values	128
CSVWriter	write comma-separated values	128
DatagramReader	read packets from the network	—
DatagramWriter	write packets to a network	—
DirectoryListing	list files in a directory keys	128
FileReader	read a file or URL	128
FileWriter	write a file	128

IO Actors		
LineReader	read lines from a file or URL	128
LineWriter	write lines to a file	128

Logic Actors		
Comparator	compare two values	112
Equals	compare n inputs for equality	112
IsPresent	determine whether a value is present	167
LogicalNot	negate a boolean	112
LogicGate	evaluate a logic function of two inputs	112
TrueGate	filter for true-valued booleans	167

Math Library		
AbsoluteValue	absolute value	57
AddSubtract	add and subtract inputs	57
Accumulator	accumulate input values	57
Average	average input values	57
Counter	count up and down	57
Differential	difference of current and previous value	57
DotProduct	dot product of array inputs	57
Limiter	limit values to a range	57
LookupTable	look up values in a table	57
Maximum	maximum of a set of inputs	58
Minimum	minimum of a set of inputs	58
MovingAverage	average of n recent inputs	58
MultiplyDivide	multiply and divide inputs	58
Quantizer	quantize the input	58
Remainder	output the remainder	58
RunningMaximum	maximum of n recent inputs	58
RunningMinimum	minimum of n recent inputs	58
Scale	multiply by a constant	58
TrigFunction	trigonometric functions	58
UnaryMathFunction	exponential, log, etc.	58

Matrix Actors		
ArrayToMatrix	convert an array to a matrix	86
MatrixJoin	join matrices by tiling	—

Matrix Actors		
MatrixSplit	split a matrix by tiles	—
MatrixToArray	convert a matrix to an array	86
MatrixToSequence	convert a matrix to a sequence of elements	—
MatrixViewer	display a matrix	—
SequenceToMatrix	convert a sequence to a matrix	—
SubMatrix	extract a submatrix	—

Random Number Generators		
Bernoulli	random true or false	114
ColtBeta	Beta random number	—
ColtBinomial	Binomial random number	—
ColtBinomialSelector	Binomial selection random number	—
ColtBreitWigner	Breit-Wigner random number	—
ColtChiSquare	Chi-squared random number	—
ColtExponential	exponential random number	248
ColtExponentialPower	exponential power random number	—
ColtGamma	gamma random number	—
ColtHyperGeometric	hyper-geometric random number	—
ColtLogarithmic	logarithmic random number	—
ColtNegativeBinomial	negative binomial random number	—
ColtNormal	normal random number	—
ColtPoisson	Poisson random number	—
ColtPoissonSlow	Poisson slow random number	—
ColtStudentT	student T random number	—
ColtVonMises	VonMises random number	—
ColtZeta	zeta random number	—
DiscreteRandomSource	discrete random number	—
Gaussian	Gaussian random number	62
RandomInteger	random 32-bit integer	—
Rician	Rician random number	—
Triangular	random number with a triangular distribution	—
Uniform	random number with a uniform distribution	147

Real-Time Actors		
DelayStart	delay start of execution of a model	—
ExecutionTime	simulate execution time	—
RealTimePlotter	plot a signal vs. real time	602
Sleep	sleep the calling thread	147

Real-Time Actors		
VariableSleep	sleep the calling thread	—
WallClockTime	output the current time of day	242

Signal Processing		
AudioCapture	capture audio from a microphone	98
AudioReader	output sampled audio from a file	—
AudioPlayer	play input audio samples	—
AudioWriter	write audio samples to an audio file	—
Autocorrelation	estimate the autocorrelation of the input	—
ClipPlayer	play an audio clip on each firing	—
ComputeHistogram	compute a histogram	601
ConvolutionalCoder	encode a binary sequence	—
DB	convert values to decibels	—
DelayLine	output arrays with a sliding window	107
DeScrambler	unrandomize a binary sequence	—
DownSample	downsample a signal	106
FFT	fast Fourier transform	101
FIR	finite impulse response filter	107
GradientAdaptiveLattice	gradient adaptive lattice filter	—
HadamardCode	produce a Hadamard code	—
HammingCoder	encode a binary sequence	—
HammingDecoder	decode a binary sequence	—
HuffmanCoder	encode a sequence with a Huffman code	—
HuffmanDecoder	decode a sequence with a Huffman code	—
IFFT	inverse fast Fourier transform	101
IIR	infinite impulse response filter	107
Lattice	FIR filter with a lattice structure	—
LempelZivCoder	encode a sequence with Lempel Ziv	—
LempelZivDecoder	decode a sequence with Lempel Ziv	—
LevinsonDurbin	Levinson-Durbin spectral estimation	—
LinearDifferenceEquationSystem	linear difference equation filter	—
LineCoder	binary sequence to symbol sequence	—
LMSAdaptive	least mean square adaptive filter	107
MaximumEntropySpectrum	maximum entropy spectral estimation	101
PhaseUnwrap	phase unwrap algorithm	—
PowerEstimate	estimate the power of the input	—
RaisedCosine	raised cosine frequency response filter	—
RecursiveLattice	IIR filter with a lattice structure	—
Scrambler	randomize a binary sequence	—

Signal Processing		
SmoothedPeriodogram	smoothed periodogram spectrum	101
Spectrum	discrete Fourier transform spectrum	101
TrellisDecoder	decode a trellis code	—
UpSample	upsample a sequence	106
VariableFIR	FIR filter with time-varying taps	107
VariableLattice	lattice filter, time-varying taps	—
VariableRecursiveLattice	recursive lattice filter, time-varying taps	—
ViterbiDecoder	decode with a Viterbi decoder	—

String Library		
StringCompare	compare two strings	125
StringFunction	trim or convert case of a string	125
StringIndexOf	find a string within another	125
StringLength	output the length of a string	125
StringMatches	compare a string against a pattern	125
StringReplace	replace a substring	125
StringSplit	split a string into pieces	125
StringSubstring	extract a substring	125

Domain Specific - Continuous		
BandlimitedNoise	bandlimited noise	326
ContinuousClock	continuous clock	326
ContinuousSinewave	continuous sine wave	326
ContinuousSpectrum	spectrum of a continuous signal	—
ContinuousTransferFunction	linear continuous filter	320
Derivative	approximate derivative	327
DifferentialSystem	nonlinear continuous system	327
DiscreteClock	discrete clock	241
Integrator	continuous integrator	316
LevelCrossingDetector	detect a level crossing	341
LinearStateSpace	linear continuous filter	327
Noise	continuous noise	—
PeriodicSampler	sample at a regular rate	341
ResettableTimer	resettable timer	241
Sampler	sample on demand	244
SingleEvent	single event	241
Waveform	continuous waveform	326
ZeroOrderHold	discrete to continuous converter	338

DomainSpecific - Discrete Event		
AverageOverTime	average taking into account time	—
Derivative	approximate derivative	327
EventFilter	filter true-valued events	244
Inhibit	conditionally block events	244
Integrator	discrete approximation to an integral	—
Merge	merge streams of events in temporal order	244
MicrostepDelay	delay by one microstep	243
MostRecent	trigger the most recently received value	244
PID	proportional, integral, derivative controller	—
Previous	output the previous event	243
Queue	queue	246
Register	latch values and produce on demand	244
ResettableTimer	resettable timer	241
Server	queue and server	246
SharedQueue	shared queue	246
SingleEvent	output a single event	241
TimeCompare	compare times of two events	242
TimeGap	output time between events on a stream	242
WaitingTime	output the time one event waits for another	242

DomainSpecific - Process Networks		
NondeterministicMerge	nondeterministically merge streams	143
OrderedMerge	merge two numerically increasing streams	143
Starver	pass a finite number of tokens	143

DomainSpecific - Rendezvous		
Barrier	barrier synchronization	146
Buffer	buffer for communication	146
Merge	merge	244
ResourcePool	pool of resources	146

DomainSpecific - Synchronous Reactive		
Absent	assert absent output	167
Current	output most recent non-absent input	167
Default	merge two streams with priority	167
EnabledComposite	composite with conditional firing	175

DomainSpecific - Synchronous Reactive		
InstantaneousDialogGenerator	test for instantaneous dialogs	—
IsPresent	true output on present input	167
NonStrictDelay	one-tick delay	167
NonStrictDisplay	display that shows absent inputs	203
NonStrictLogicGate	parallel or and and	169
NonStrictThreeBitAdder	test actor	—
When	gate a signal with another	167
TrueGate	gate a signal	167

Although they are not part of the Ptolemy II standard library, the Ptolemy distribution includes a collection of useful actors in the `MoreLibraries` library. The table below highlights a few of these.

More Libraries		
ArrayOfRecordsRecorder	display an array of records	—
CalInterpreter	actor defined in the Cal actor language	—
DatabaseInsert	database insert via a DatabaseManager	—
DatabaseManager	interface to a database	—
DatabaseQuery	perform a query via a DatabaseManager	—
DatabaseSelect	perform a select via a DatabaseManager	—
Exec	execute a command-line statement	—
HttpActor	react to HTTP requests via a WebServer actor	584
JSONToToken	convert a JSON string to a record	—
KeyWriter	write a key out to a key store	—
SQLStatement	issue an SQL statement via a DatabaseManager	—
FSMActor	finite state machine without hierarchy	188
MatlabExpression	compute an output using a MATLAB script	437
NonStrictTest	test inputs against expected values	126
PrivateKeyReader	produce a private key from a key store	—
PublicKeyReader	produce a public key from a key store	—
PythonActor	actor specified in the Python language	437
PythonScript	actor specified in the Python language	437
SecretKey	create a secret key	—
SecretKeyReader	produce a secret key from a key store	—
SendMail	send email	—
SerialComm	read from or write to a serial port	—
SignatureSigner	sign data using a private key	—
SignatureVerifier	verify signed data	—
Simulator	run a program with socket communication	—

More Libraries		
StringToXML	convert XML-formatted string to an XML token	634
SymmetricDecryption	decrypt data	—
SymmetricEncryption	encrypt data	—
SystemCommand	invoke an external program and report results	—
Test	test inputs against expected values	126
TestExceptionAttribute	check exceptions against an expected exception	126
TokenToJSON	convert a record to a JSON string	—
WebServer	start a web server on the local machine	584
XMLInclusion	combine XML tokens into one	—
XSLTransformer	transform XML using XSLT	—