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.

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.
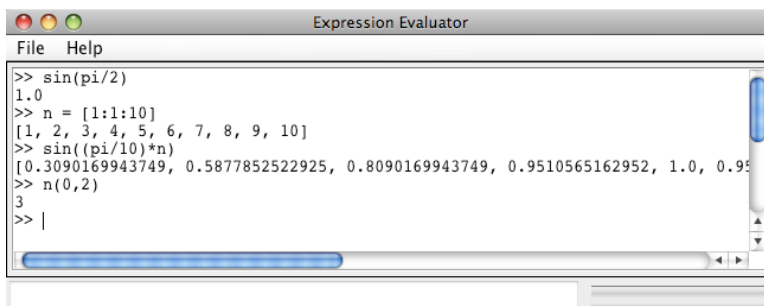

Figure 13.1: The Expression Evaluator.

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	-9223372036854775808L	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, counterintuitively,

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

When an operator involves two distinct types, the expression language has to make a decision about which type to use to implement the operation. If one of the two types can be converted without loss into the other, then it will be. For instance, *int* can be converted losslessly to *double*, so 1.0/2 will result in 2 being first converted to 2.0, so the result will be 0.5. Among the scalar types, *unsignedByte* can be converted to anything else, *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 "noise-Level." 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.
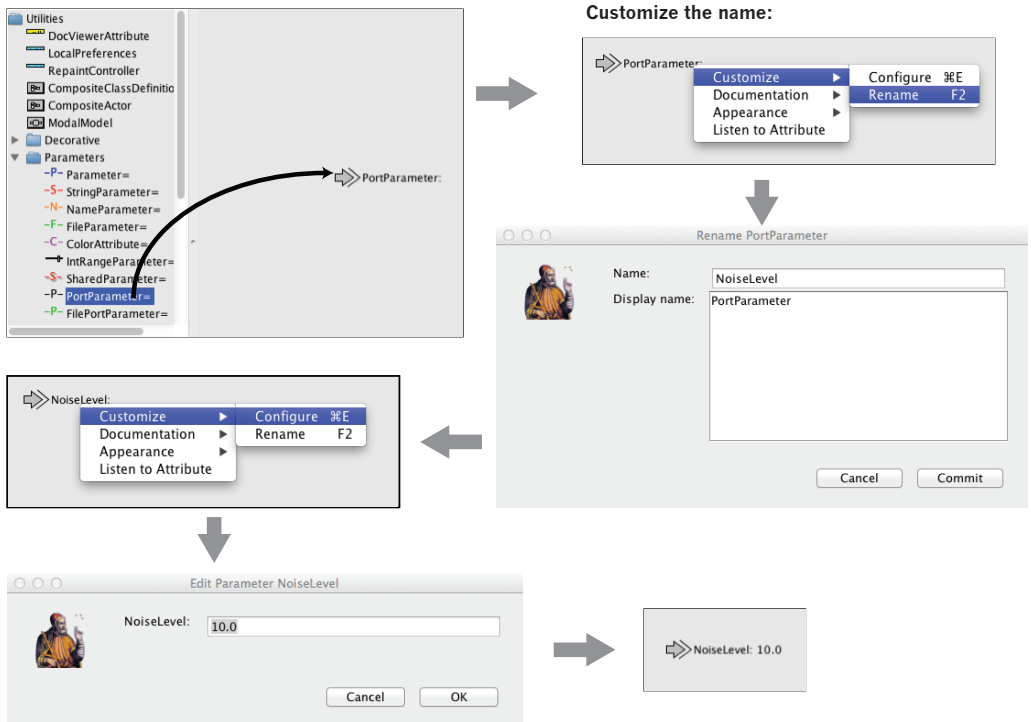

Figure 13.2: A *PortParameter* is both a port and a parameter. To use it in a composite actor, drag it into the actor, change its name to something meaningful and set its default value.
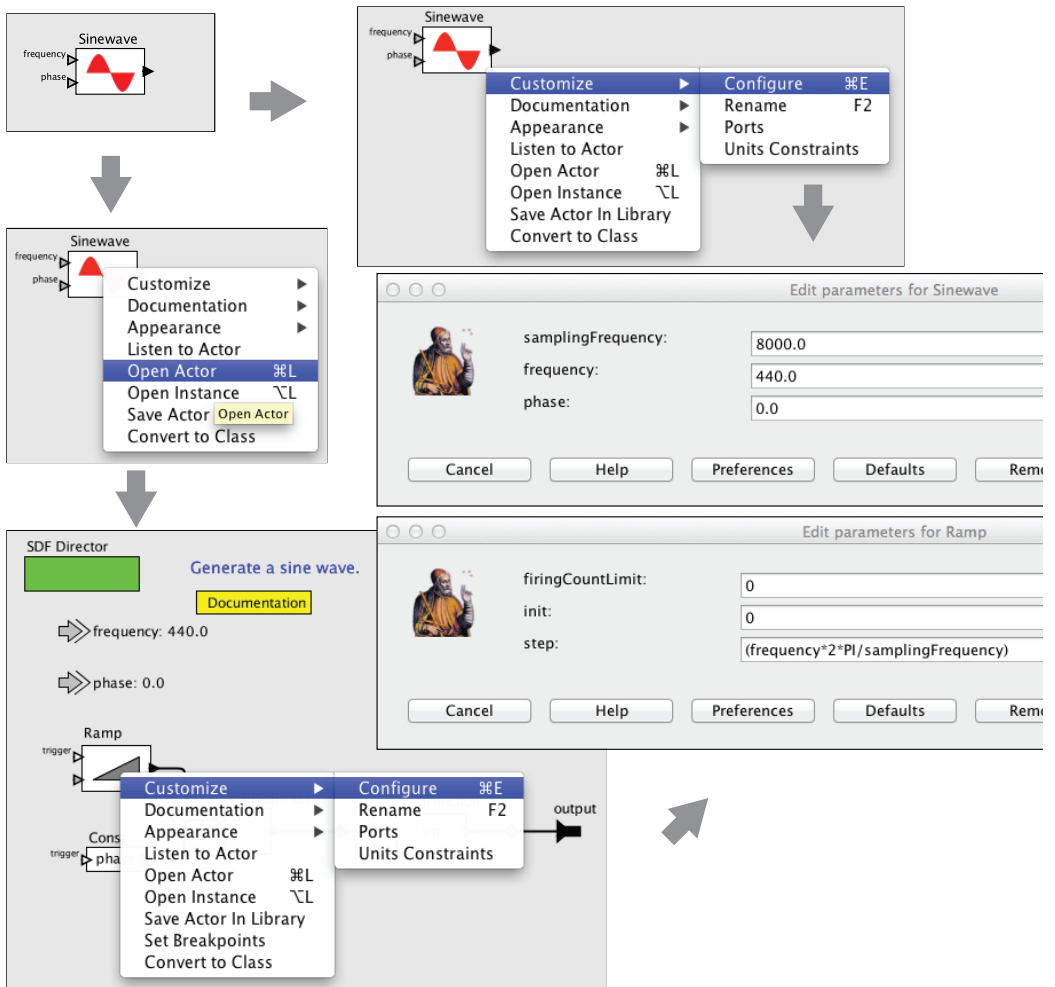
Figure 13.3: Sinewave actor, showing its port parameters, and their use at the lower level of hierarchy.
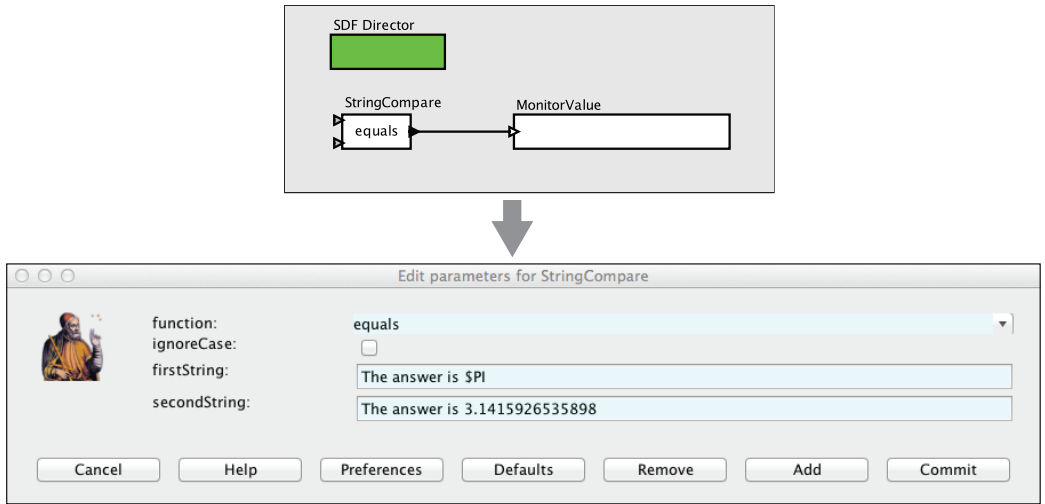
Figure 13.4: String parameters are indicated in the parameter editor boxes by a light blue background. A string parameter can include references to variables in scope with $*name*, where *name* is the name of the variable. In this example, the built-in constant $PI is referenced by name in the first 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.
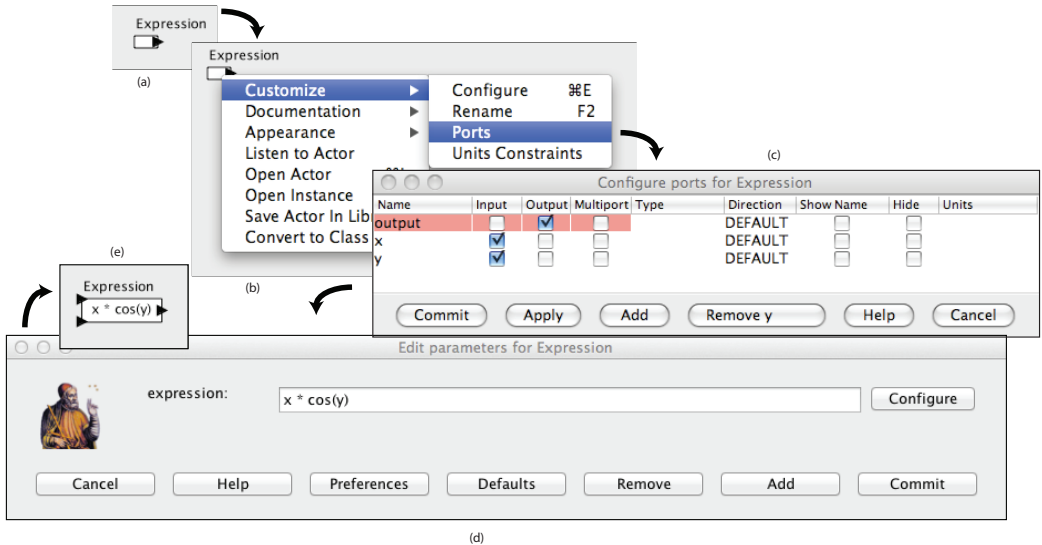
Figure 13.5: Illustration of the Expression actor.

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 *unsigned-Byte* 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.
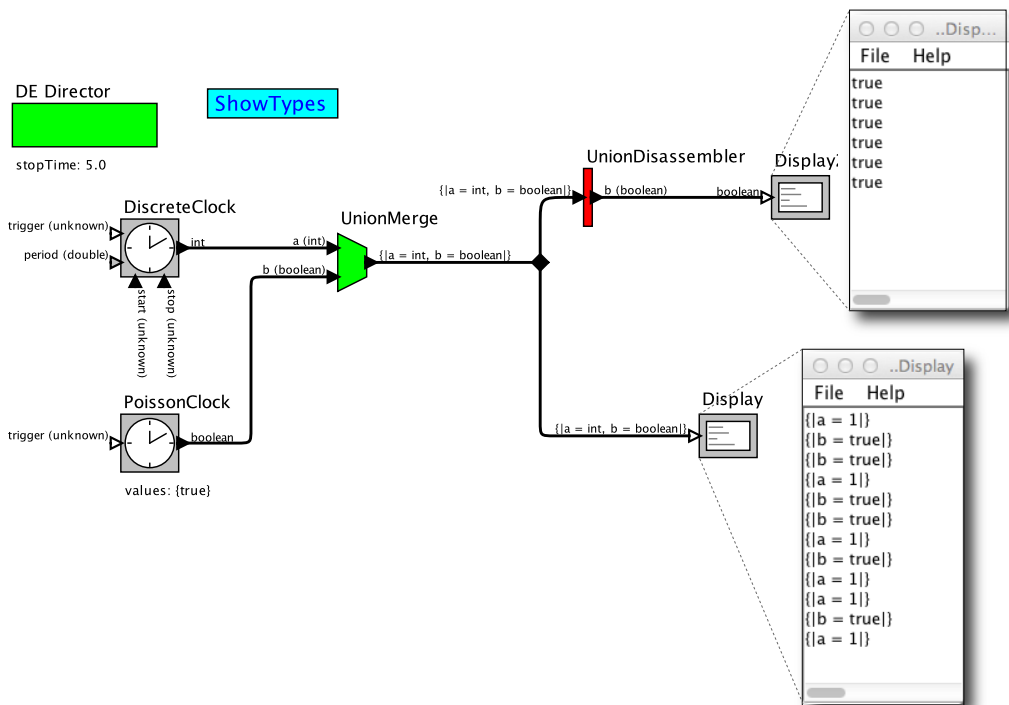

Figure 13.6: Union types allow types to resolve to more than one type. [online]

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 param-
eter.

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.

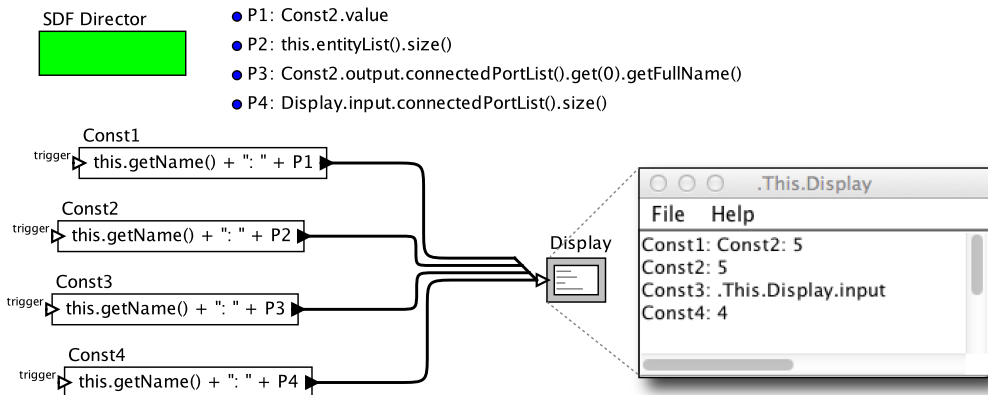Now, the first two outputs of this model should be easy to understand:

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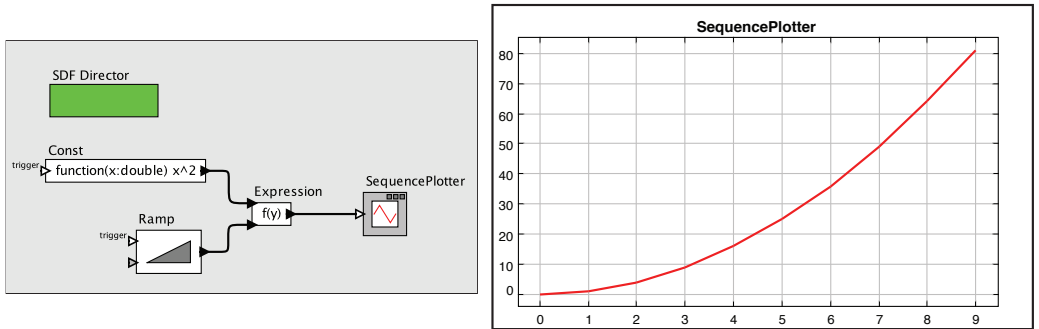
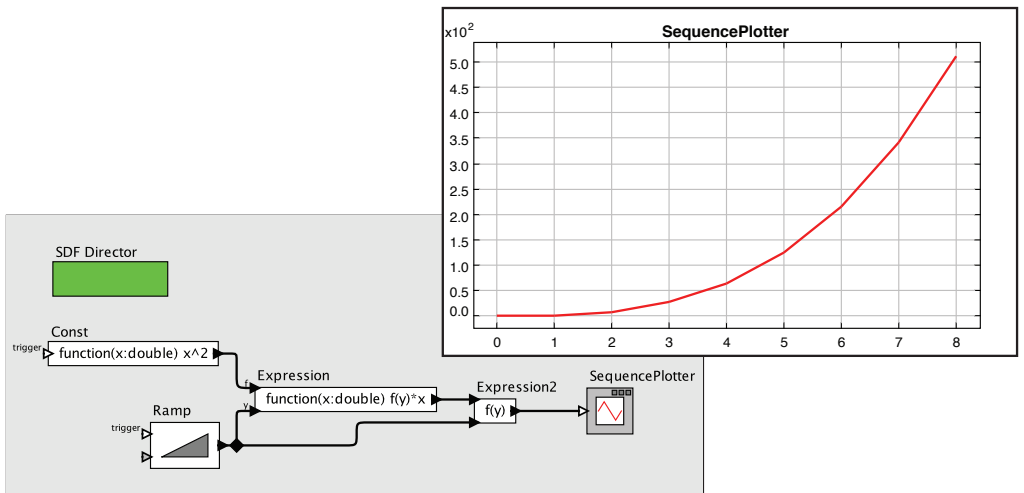Figure 13.8: Example of a function being passed from one actor to another.

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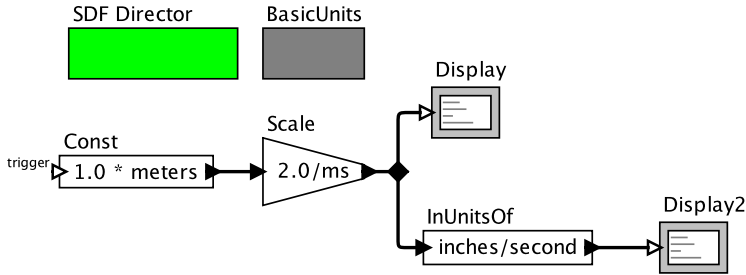
Figure 13.10: Example of a model that includes a unit system. [online]

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.
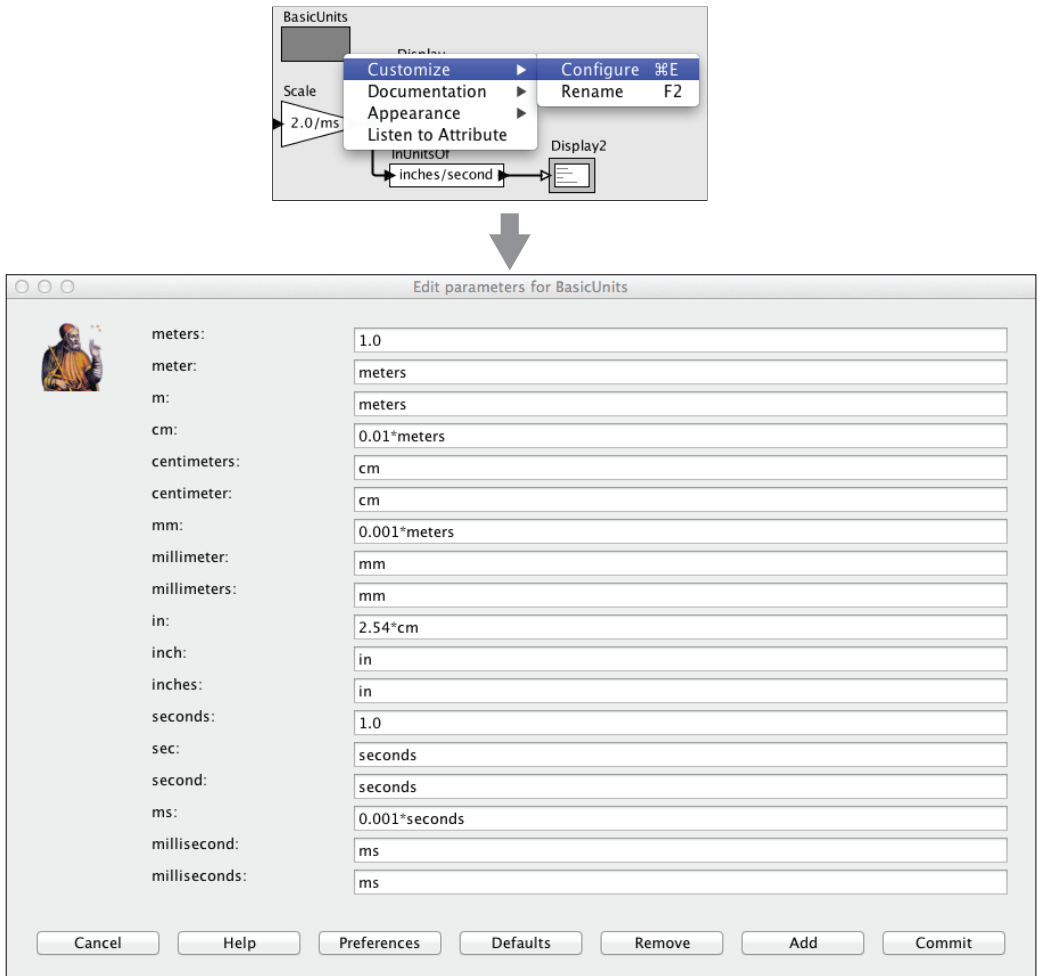
Figure 13.11: Units defined in a units system can be examined by double clicking or by right clicking and selecting Customize and Configure.
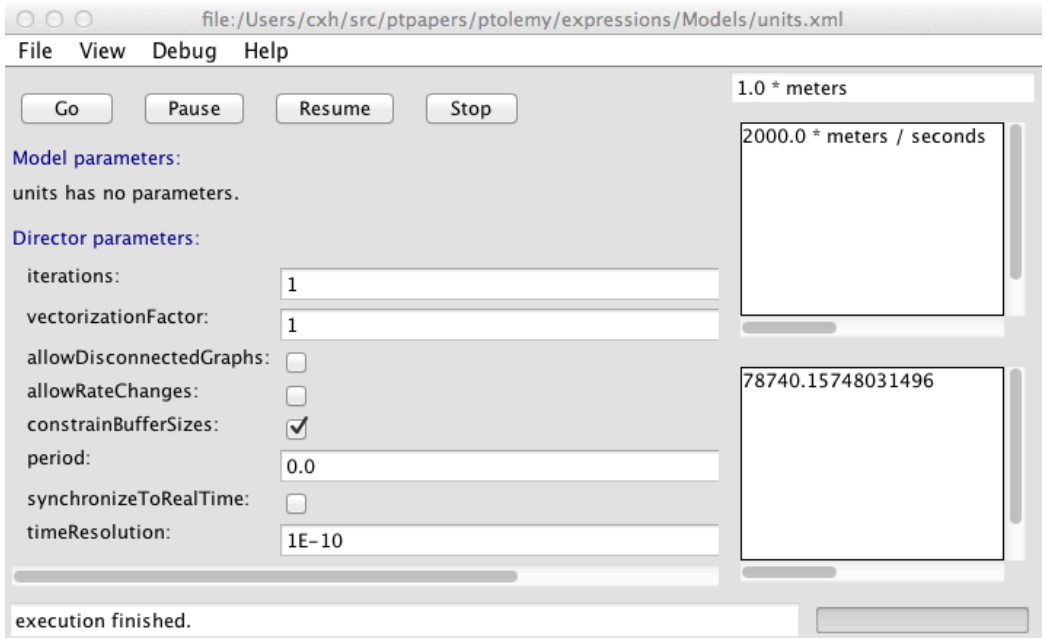
Figure 13.12: Result of running the model in Figure 13.10.

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: $atan(z) = -\frac{i}{2}\log\left(\frac{i-z}{i+z}\right)$
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: $acosh(z) = \log\left(z + \sqrt{z^2 - 1}\right)$
asinh	*double* or *complex*	*double* or *complex*	hyperbolic arc sine *complex* case: $asinh(z) = \log\left(z + \sqrt{z^2 + 1}\right)$
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 *array-Type(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: $sqrt(z) = \sqrt{|z|}(\cos\left(\frac{\angle z}{2}\right) + i\sin\left(\frac{\angle z}{2}\right))$
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 *array-Type(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: $w(n) = 0.42 + 0.5\cos\left(\frac{2\pi n}{M}\right) + 0.08\cos\left(\frac{4\pi n}{M}\right)$
generateBlackman HarrisWindow	*int*	*arrayType(double)*	Return a Blackman-Harris window with the specified length. For length M + 1, the formula is: $w(n) = 0.35875 + 0.48829\cos\left(\frac{2\pi n}{M}\right) + 0.14128\cos\left(\frac{4\pi n}{M}\right) + 0.01168\cos\left(\frac{6\pi n}{M}\right)$
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: $w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M}\right)$
generateHanning Window	*int*	*arrayType(double)*	Return a Hanning window with the specified length. For length M + 1, the formula is: $w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M}\,M\right)$
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: $h(t) = \frac{\sin\left(\frac{\pi t}{T}\right)}{\frac{\pi t}{T}} \times \frac{\cos\left(\frac{x\pi t}{T}\right)}{1-\left(\frac{2xt}{T}\right)^2}$, 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.