Chapter 3 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture) ," Technical Memorandum UCB/ERL M04/16, University of California, Berkeley, CA USA 94720, June 24, 2004.

3

Data Package

Authors: *Rowland R. Johnson*
Bart Kienhuis
Edward A. Lee
Xiaojun Liu
Steve Neuendorffer
Neil Smyth
Yuhong Xiong

3.1 Introduction

The data package provides data encapsulation, polymorphism, parameter handling, an expression language, and a type system. Figure 3.1 shows the key classes in the main package (subpackages will be discussed later).

3.2 Data Encapsulation

The Token class and its derived classes encapsulate application data. Tokens can be transported via message passing between Ptolemy II objects, and can be used to parameterize Ptolemy II actors. Encapsulating data in this way provides a standard interface so that data can be handled uniformly regardless of its detailed structure. Such encapsulation allows for a great degree of extensibility, permitting developers to extend the library of data types that Ptolemy II can handle. It also permits a user interface to interact with application data without detailed prior knowledge of the structure of the data.

Token classes are provided for encapsulating many different types of data, such as integers (IntToken), double precision floating point numbers (DoubleToken), and complex numbers (ComplexToken). A special Token subclass (EventToken) exists for representing the presence of a “pure event” that encapsulates no data. Tokens can encapsulate data structures of arbitrary size. All data tokens share several properties, including immutability, type-polymorphic operations, and the possibility for automatic type conversions. These properties will be described in later sections.

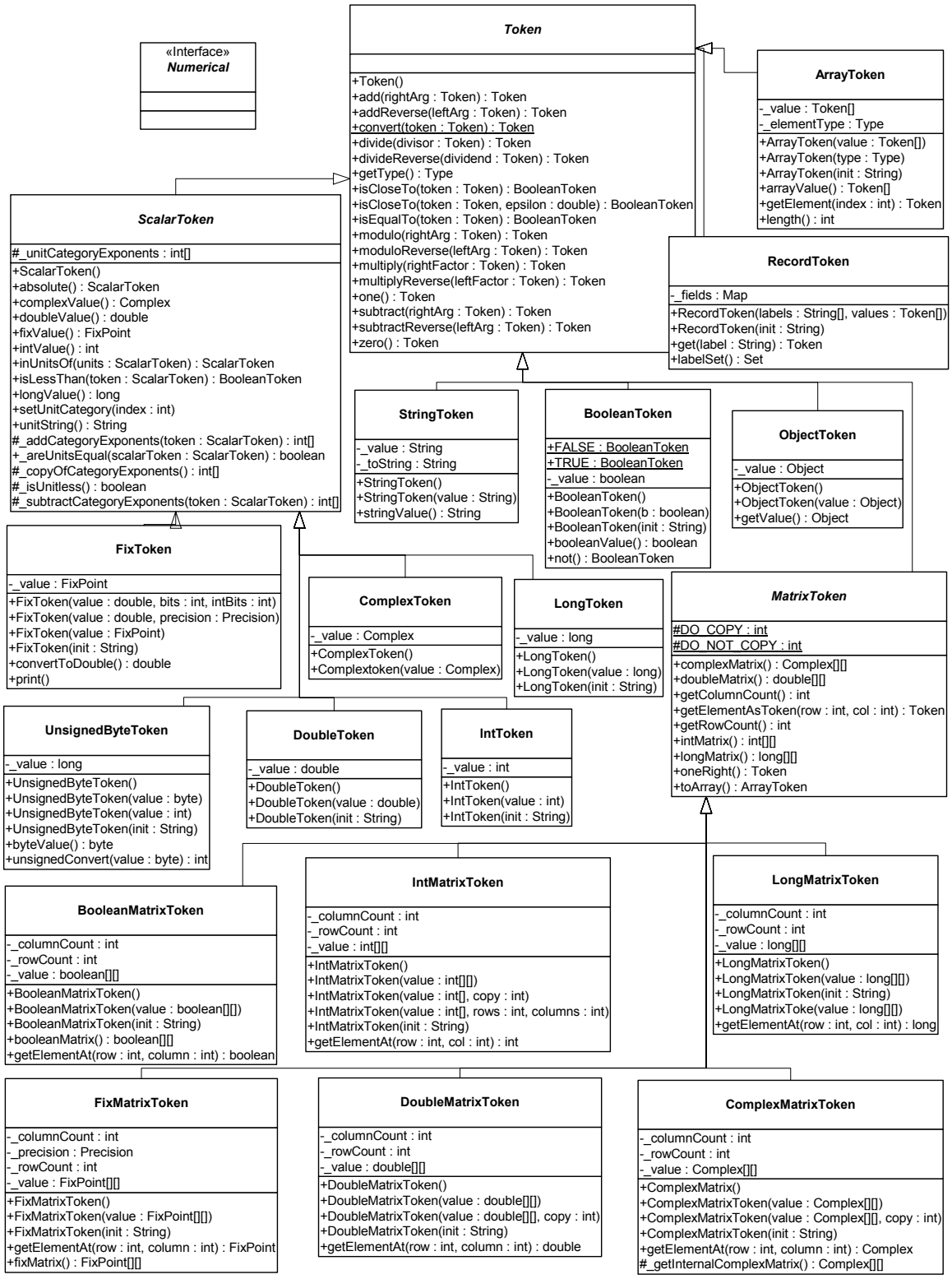


FIGURE 3.1. Static Structure Diagram (Class Diagram) for the classes in the data package.

3.2.1 Matrix data types

The MatrixToken base class provides basic structure for two-dimensional arrays of data. Various derived classes encapsulate data of different types, such as integers, and complex numbers. Standard matrix-matrix and scalar-matrix operations are defined.

3.2.2 Array and Record data types

An ArrayToken is a token that contains an array of tokens. All the element tokens must have the same type, but that type is arbitrary. For instance, it is possible to constructs arrays of arrays of any type of token. The ArrayToken class differs from the various MatrixToken classes in that MatrixTokens contain only be constructed for primitive data, such as int or double, while an array can be constructed for arbitrary token types. In other words, matrix tokens are specialized for storing two dimensional structures of primitive data, while array tokens offer more flexibility in type specifications.

A RecordToken contains a set of labeled values, and operates similarly to `struct` in the C language. The values can be arbitrary tokens and are not required to have the same type.

3.2.3 Fixed Point Data Type

The FixToken class encapsulates fixed point data. The UML diagram showing classes involved in the definition of the FixPoint data type is shown in Figure 3.2. The FixToken class encapsulates an instance of the FixPoint class in the math package. The underlying FixPoint class is implemented using Java's BigInteger class to represent fixed point values. The advantage of using the BigInteger package is that it makes this FixPoint implementation truly platform independent and furthermore, it doesn't put any restrictions on the maximal number of bits allowed to represent a value.

The precision of a FixPoint data type is represented by the Precision class. This class does the parsing and validation of the various specification styles we want to support. It stores a precision into two separate integers. One number represents the number of integer bits, and the other number represents the number of fractional bits. For convenience, the precision of fixed point data can be specified in two different ways:

(m/n): The total precision of the output is m bits, with the integer part having n bits. The fractional part thus has $m - n$ bits.

($m.n$): The total precision of the output is $n + m$ bits, with the integer part having m bits, and the fractional part having n bits.

The Quantization class represents various quantization techniques. Creating a FixPoint value requires specifying a double value and an instance of the Quantization class. For convenience, static methods are provided in the Quantizer class that create FixPoint instances without referencing a Quantization explicitly. During conversion, the handling of overflow and underflow is handled by specifying instances of the Overflow class.

The convertToDouble() method in the FixToken class converts a fixed point value into a double representation. Note that the getDouble() method defined by Token is not used since conversion from a FixPoint to a double is not, in general, a lossless conversion and is hence not allowed automatically. For details about how to represent Fixed Point numbers in the expression language, see volume 1, the Expressions chapter.

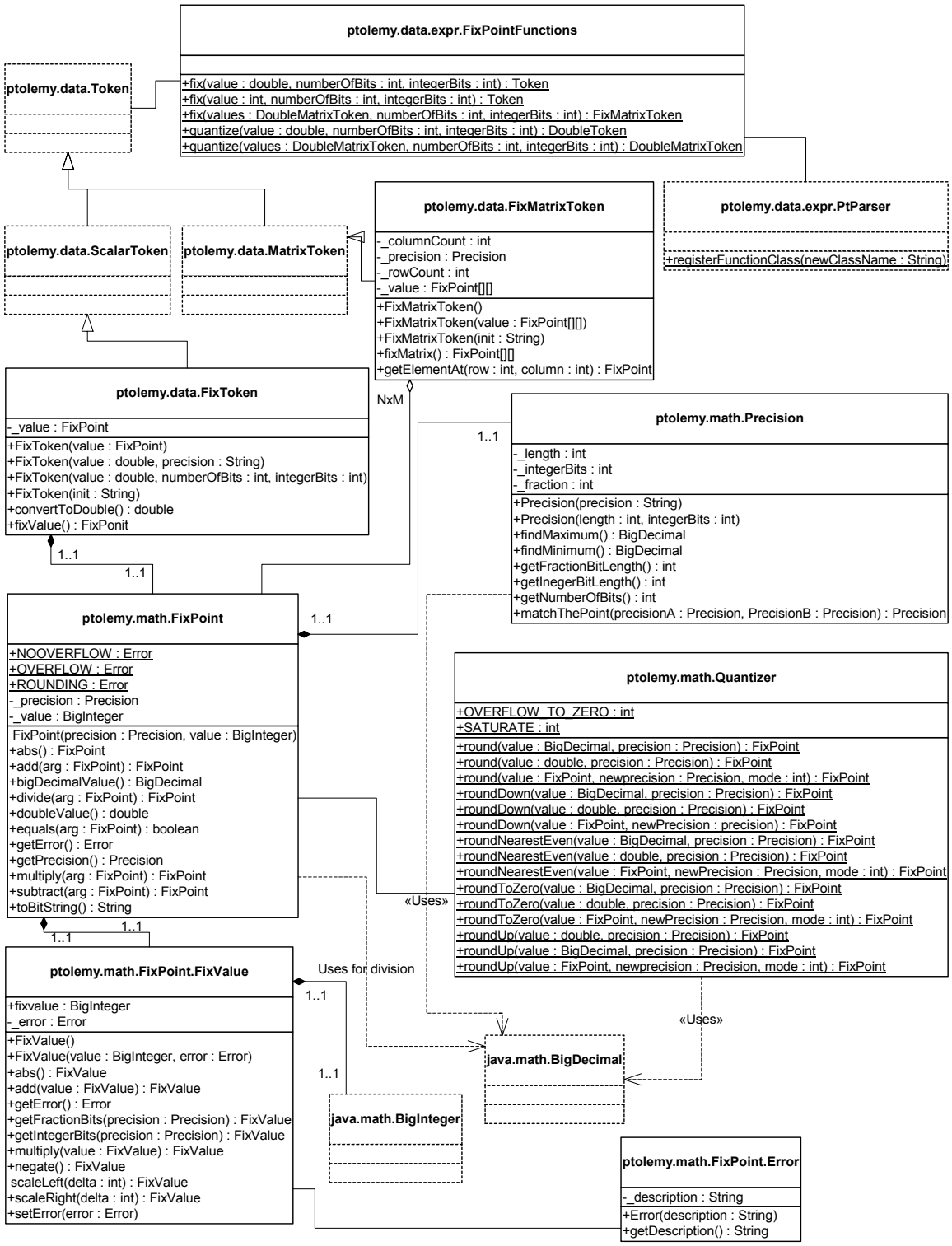


FIGURE 3.2. Organization of the FixPoint Data Type.

3.2.4 Function Closures

The FunctionToken class encapsulates functions that can be evaluated. These function closures can be passed as messages just like any other tokens. When a function closure is created, all identifiers that are not arguments to the function are evaluated. The arguments to the function, however are only evaluated when the function is applied. For information on how functions closures can be represented in the expression language, see volume 1.

3.3 Immutability

Tokens in Ptolemy II are, in general, immutable. This means that a token's value cannot be changed after the token is constructed. The value of a token must be specified by constructor arguments, and there is no other mechanism for setting the value. If a token encapsulating another value is required, a new instance of Token must be constructed.

There are several reasons for making tokens immutable.

- First, when a token is sent to several receivers, we want to be sure that all receivers get the same data. Each receiver is sent a reference to the same token. If the Token were not immutable, then it would be necessary to clone the token for all receivers after the first one.
- Second, since a token is passed between two actors, they may both have a reference to the token. If the token were mutable, then the token would represent shared state of the two actors, requiring synchronization and limiting the ability to represent distributed computation. Immutable tokens passed between actors ensures that the concurrency of actors is determined solely by a model of computation.
- Third, we use tokens to parameterize actors, and parameters often have mutual dependencies. That is, the value of a parameter may depend on the value of other parameters. The value of a parameter is represented by an instance of Token. If that token were allowed to change value without notifying the parameter, then the parameter would not be able to notify other parameters that depend on its value. Thus, a mutable token would have to implement a publish-and-subscribe mechanism so that parameters could subscribe and thus be notified of any changes. By making tokens immutable, we greatly simplify the design.
- Finally, having our Tokens immutable makes them similar in concept to the data wrappers in Java, like Double, Integer, etc., which are also immutable.

In most cases, the immutability of tokens is enforced by the design of the Token subclasses. One exception is the ObjectToken class. An ObjectToken contains a reference to an arbitrary Java object created by the user, and a reference to this object can be retrieve through the getValue() method. Since the user may modify the object after the token is constructed, the immutability of an ObjectToken is difficult to ensure. Although it could be possible to clone the object in the ObjectToken constructor and return another clone in the getValue() method, this would require the object to be cloneable, severely limiting the use of the ObjectToken. In addition, since the default implementation of clone() only makes a shallow copy, this approach is not able to enforce immutability on all cloneable objects. Cloning a large object could be prohibitively expensive. For these reasons, the ObjectToken does not attempt to enforce immutability, but rather relies on the cooperation from the user. Violating this convention could lead to unintended non-determinism.

For matrix tokens, enforced immutability requires the contained matrix (Java array) to be copied when the token is constructed and when the matrix is returned in response to queries such as intMatrix(), doubleMatrix(), etc. Since the cost of copying large arrays is non-trivial, the user should not

make more queries than necessary. For optimization, some matrix token classes have a constructor that takes a flag, which specifies whether the given array needs to be copied or not. The getElementAt() method can be used to read the contents of the matrix without copying the internal array.

3.4 Polymorphism

3.4.1 Polymorphic Arithmetic Operators

One of the goals of the data package is to support polymorphic operations between tokens. For this, the base Token class defines methods for primitive arithmetic operations, which are add(), multiply(), subtract(), divide(), modulo() and equals(). Derived classes override these methods to provide class specific operation where appropriate. The objective here is to be able to say, for example,

```
a.add(b)
```

where `a` and `b` are arbitrary tokens. If the operation `a + b` makes sense for the particular tokens, then the operation is carried out and a token of the appropriate type is returned. If the operation does not make sense, then an exception is thrown. Consider the following example

```
IntToken a = new IntToken(5);
DoubleToken b = new DoubleToken(2.2);
StringToken c = new StringToken("hello");
```

then

```
a.add(b)
```

gives a new DoubleToken with value 7.2,

```
a.add(c)
```

gives a new StringToken with value "5Hello", and

```
a.modulo(c)
```

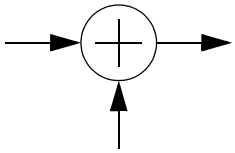
throws an exception. Thus in effect we have overloaded the operators +, -, *, /, %, and ==.

It is not always immediately obvious what is the correct implementation of an operation and what the return type should be. For example, the result of adding an integer token to a double-precision floating-point token should probably be a double, not an integer. The mechanism for making such decisions depends on a *type hierarchy* that is defined separately from the class hierarchy. This type hierarchy is explained below.

The token classes also implement the methods zero() and one() which return the additive and multiplicative identities respectively. These methods are overridden so that each token type returns a token of its type with the appropriate value. For matrix tokens, zero() returns a zero matrix whose dimension is the same as the matrix of the token where this method is called; and one() returns the left identity, i.e., it returns an identity matrix whose dimension is the same as the number of rows of the matrix of the token. Another method oneRight() is also provided in numerical matrix tokens, which returns the right identity, i.e., the dimension is the same as the number of columns of the matrix of the token.

Since data is transferred between entities using Tokens, it is straightforward to write polymorphic actors that receive tokens on their inputs, perform one or more of the overloaded operations and output

the result. For example an `add` actor that looks like this:



might contain code like:

```
Token input1, input2, output;
// read Tokens from the input channels into input1 and input2 variables
output = input1.add(input2);
// send the output Token to the output channel.
```

We call such actors *data polymorphic* to contrast them from *domain polymorphic* actors, which are actors that can operate in multiple domains. Of course, an actor may be both data and domain polymorphic.

3.4.2 Automatic Type Conversion

For the above arithmetic operations, if the two tokens being operated on have different types, type conversion is needed. Generally speaking, Ptolemy II automatically performs conversions that do not lose numerical precision. Other conversion must be explicitly represented by the user. The admissible automatic type conversions between different token types are modeled as a partially ordered set called the *type lattice*, shown in figure 3.3. In that diagram, type *A* is *greater than* type *B* if there is a path upwards from *B* to *A*. Thus, ComplexMatrix is greater than Int. Type *A* is *less than* type *B* if there is a path downwards from *B* to *A*. Thus, Int is less than ComplexMatrix. Otherwise, types *A* and *B* are *incomparable*. Complex and Long, for example, are incomparable. In the type lattice, a type can be automatically converted to any type greater than it.

This hierarchy is realized by the TypeLattice class in the data.type subpackage. Each node in the lattice is an instance of the Type interface. The TypeLattice class provides methods to compare two token types.

Two of the types, *Numerical* and *Scalar*, are abstract. They cannot be instantiated. This is indicated in the type lattice by italics.

Type conversion is done by the convert() method in type classes. This method converts the argument into a token with the same type. For example, BaseType.DoubleType.convert(Token token) converts the specified token into an instance of DoubleToken. The convert() method can convert any token immediately below it in the type hierarchy into an instance of its own class. If the argument is higher in the type hierarchy, or is incomparable with its own class, the convert() method throws an exception. If the argument to convert() already has the correct type, it is returned without any change. Many of the simpler token classes also provide a static convert() method that can be used more simply than the convert() method of the corresponding type.

Most implementations of the add(), subtract(), multiply(), divide(), modulo(), and equals() methods require that the type of the argument and the implementing class be comparable in the type hierarchy. If this condition is not met, these methods will throw an exception. If the type of the argument is lower than the type of the implementing class, then the argument is usually converted to the type of the implementing class before the operation is carried out. One exception is the implementation of these

methods for matrix tokens. To allow matrices to be multiplied and divided by scalars, the normal conversion is not performed. The MatrixToken base class deals specially with scalar-matrix operations.

To allow this, the implementation of most operations is somewhat more complicated if the type of the method argument is higher than the implementing class. In this case, we assume the operation is implemented in the class that has the higher type (the matrix token in the above example). Since token operations need not be commutative, for example, `"Hello" + "world"` is not the same as `"world" + "Hello"`, and 3-2 is not the same as 2-3, the implementation of arithmetic operations cannot simply call the same method on the class of the argument. Instead, a separate set of methods is provided, which perform token operations in the reverse order. These methods are addReverse(), subtractReverse(), multiplyReverse(), divideReverse(), and moduloReverse(). The equality check is always commutative so no equalsReverse() is needed. Under this setup, a.add(b) means $a+b$, and a.addReverse(b) means $b+a$, where a and b are both tokens. If, for example, when a.add(b) is invoked and the type of b is higher than a, the add() method of a will automatically call b.addReverse(a) to carry out the addition.

For scalar and matrix tokens, methods are also provided to convert the content of the token into another numeric type. In the ScalarToken base class, these methods are intValue(), longValue(), doubleValue(), fixValue(), and complexValue(). In the MatrixToken base class, the methods are intMatrix(),

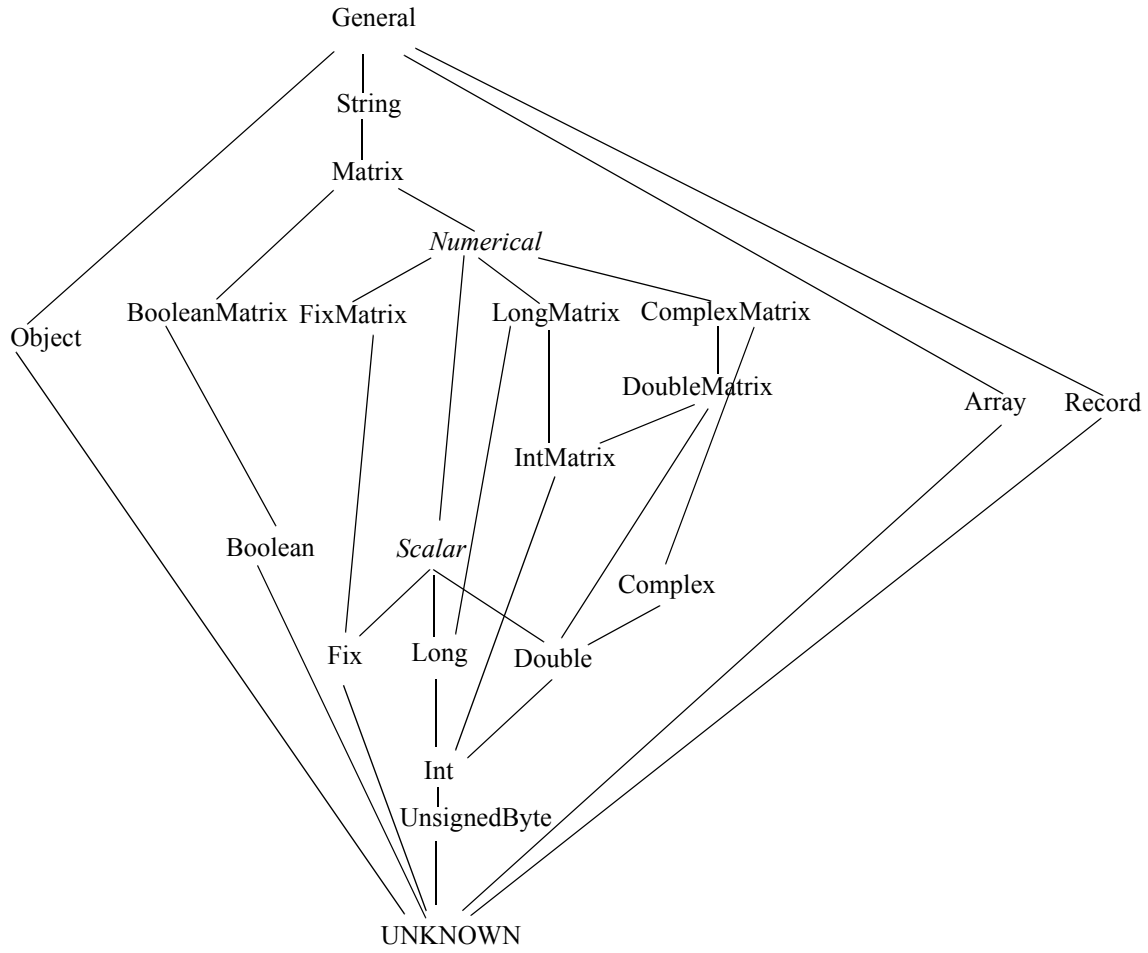


FIGURE 3.3. The type lattice.

longMatrix(), doubleMatrix(), fixMatrix(), and complexMatrix(). The default implementation in these two base classes simply throws an exception. Derived classes override these methods according to the automatic type conversion relation of the type lattice. For example, the IntToken class overrides all the methods defined in ScalarToken, but the DoubleToken class does not override the intValue() method, since automatic conversion is not allowed from a double to an integer.

3.5 Variables and Parameters

In Ptolemy II, any instance of NamedObj can have attributes, which are instances of the Attribute class. A *variable* is an attribute that contains a token. Its value can be specified by an expression that can refer to other variables. A *parameter,* implemented by the Parameter class, is in most ways functionally identical to a variable, but also appears modifiable from the user interface. See figure 3.4 and figure 3.5. The presence of these two separate classes allows variables to exist which are internal to an actor, and not visible to an end user. For the rest of this section we consider parameters and variables to be largely interchangeable.

3.5.1 Values

The value of a variable can be specified by a token passed to a constructor, a token set using the setToken() method, or an expression set using the setExpression() method.

When the value of a variable is set by setExpression(), the expression is not actually evaluated until you call getToken() or getType(). This is important, because it implies that a set of interrelated expressions can be specified in any order. Consider for example the sequence:

```
Variable v3 = new Variable(container,"v3");
Variable v2 = new Variable(container,"v2");
Variable v1 = new Variable(container,"v1");
v3.setExpression("v1 + v2");
v2.setExpression("1.0");
v1.setExpression("2.0");
v3.getToken();
```

Notice that the expression for `v3` cannot be evaluated when it is set because `v2` and `v1` do not yet have values. But there is no problem because the expression is not evaluated until getToken() is called. Obviously, an expression can only reference variables that are added to the scope of this variable before the expression is evaluated (i.e., before getToken() is called). Otherwise, getToken() will throw an exception. By default, all variables contained by the same container or any container above in the hierarchy are in the scope of this variable. Thus, in the example above, all three variables are in each other's scope because they belong to the same container. This is why the expression `"v1 + v2"` can be evaluated. If two containers above in the hierarchy contain the same variable, then the one lowest in the hierarchy will shadow the one that is higher. That is, the lower one will be used to evaluate the expression.

3.5.2 Types

Ptolemy II, in contrast to Ptolemy Classic, does not have a plethora of type-specific parameter classes. Instead, a parameter has a type that reflects the token it contains. The allowable types of a

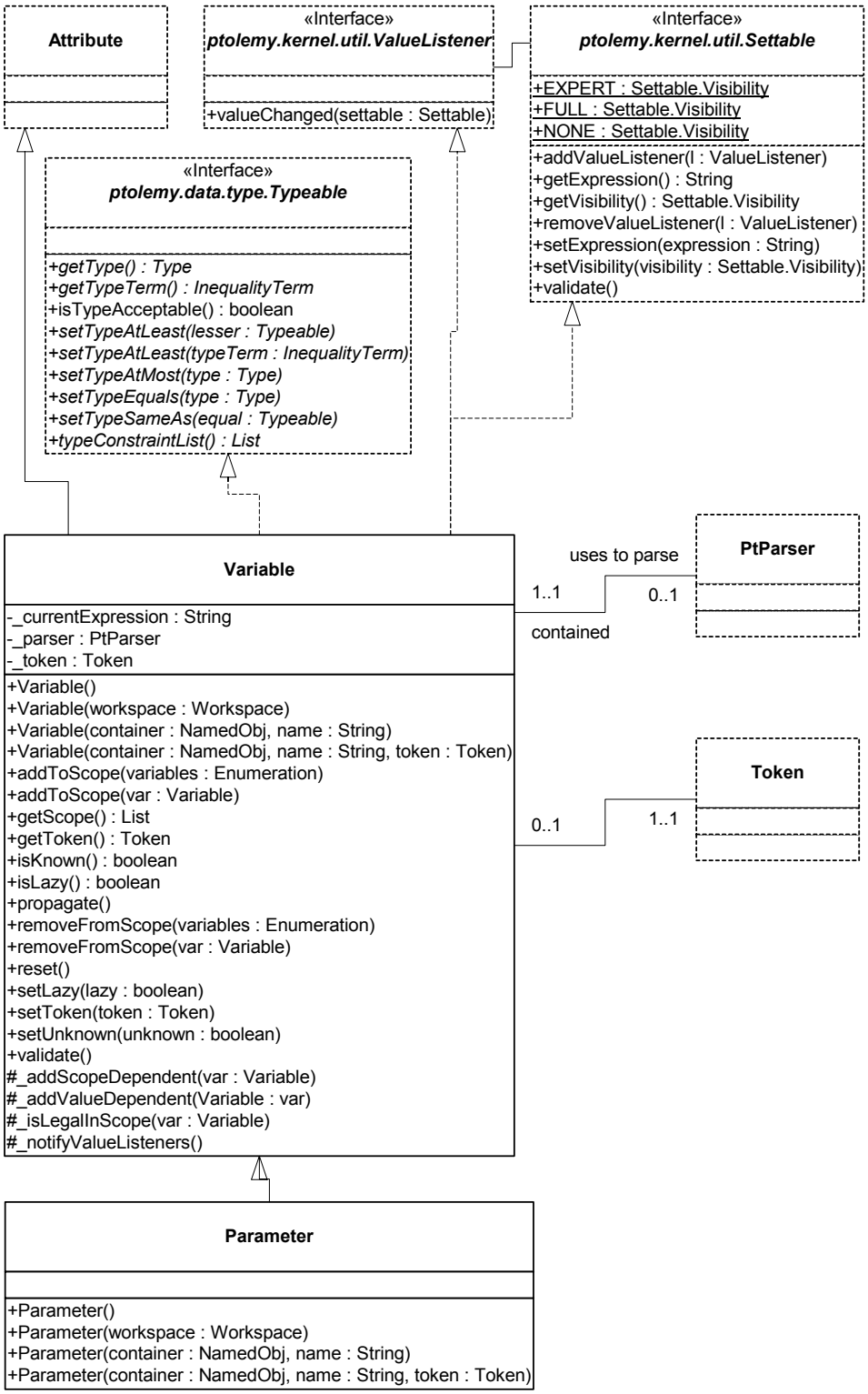


FIGURE 3.4. Static structure diagram for the Variable and Parameter classes in the data.expr package.

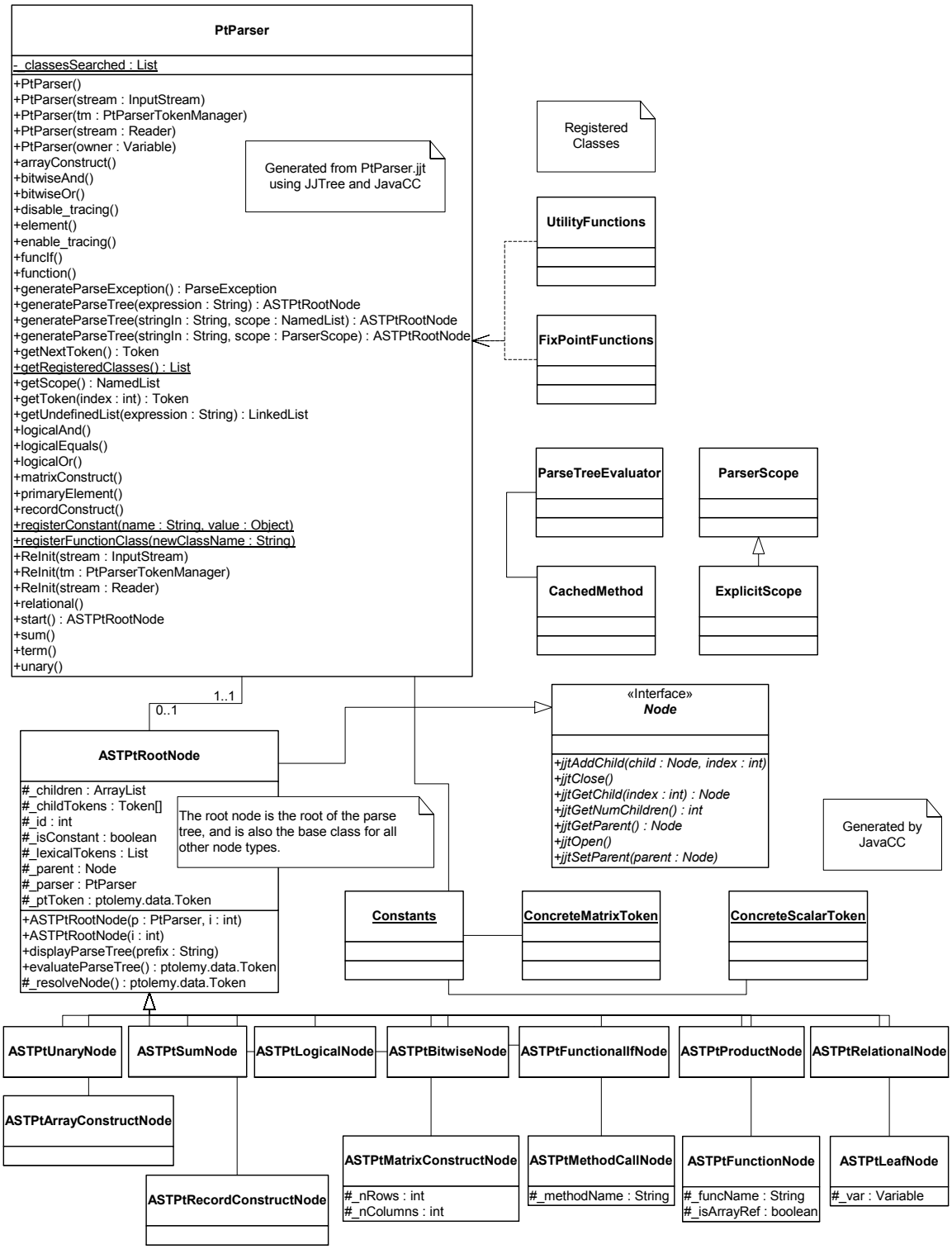


FIGURE 3.5. Static structure diagram for the parser classes in the data.expr package

parameter or variable can also be constrained using the following mechanisms:

- You can require the variable to have a specific type. Use the setTypeEquals() method.
- You can require the type to be at most some particular type in the type hierarchy (see the Type System chapter to see what this means).
- You can constrain the type to be the same as that of some other object that implements the Typeable interface.
- You can constrain the type to be at least that of some other object that implements the Typeable interface.

Except for the first type constraint, these are not checked by the Variable class. They must be checked by a type resolution algorithm, which is executed before the model runs and after parameter values change.

The type of the variable can be specified in a number of ways, all of which require the type to be consistent with the specified constraints (or an exception will be thrown):

- It can be set directly by a call to setTypeEquals(). If this call occurs after the variable has a value, then the specified type must be compatible with the value. Otherwise, an exception will be thrown. Type resolution will not change the type set through setTypeEquals() unless the argument of that call is null. If this method is not called, or called with a null argument, type resolution will resolve the variable type according to all the type constraints. Note that when calling setTypeEquals() with a non-null argument while the variable already contains a non-null token, the argument must be a type no less than the type of the contained token. To set type of the variable lower than the type of the currently contained token, setToken() must be called with a null argument before setTypeEquals().
- Setting the value of the variable to a non-null token constrains the variable type to be no less than the type of the token. This constraint will be used in type resolution, together with other constraints.
- The type is also constrained when an expression is evaluated. The variable type must be no less than the type of the token the expression is evaluated to.
- If the variable does not yet have a value, then the type of a variable may be determined by type resolution. In this case, a set of type constraints is derived from the expression of the variable (which presumably has not yet been evaluated, or the type would be already determined). Additional type constraints can be added by calls to the setTypeAtLeast() and setTypeSameAs() methods.

Subject to specified constraints, the type of a variable can be changed at any time. Some of the type constraints, however, are not verified until type resolution is done. If type resolution is not done, then these constraints are not enforced. Type resolution is normally done by the Manager that executes a model.

The type of the variable may change when setToken() or setExpression() is called.

- If no expression, token, or type has been specified for the variable, then the type becomes that of the current value being set.
- If the variable already has a type, and the value can be converted losslessly into a token of that type, then the type is left unchanged.
- If the variable already has a type, and the value cannot be converted losslessly into a token of that

type, then the type is changed to that of the current value being set.

If the type of a variable is changed after having once been set, the container is notified of this by calling its attributeTypeChanged() method. If the container does not allow type changes, it should throw an exception in this method. If the value is changed after having once been set, then the container is notified of this by calling its attributeChanged() method. If the new value is unacceptable to the container, it should throw an exception. The old value will be restored.

The token returned by getToken() is always of the type given by the getType() method. This is not necessarily the same as the type of the token that was inserted via setToken(). It might be a distinct type if the contained token can be converted losslessly into one of the type given by getType(). In rare circumstances, you may need to directly access the contained token without any conversion occurring. To do this, use getContainedToken().

3.5.3 Dependencies

Expressions set by setExpression() can reference any other variable that is within scope. By default, the scope includes all variables contained by the same container or any container above it in the hierarchy. In addition, any variable can be explicitly added to the scope of a variable by calling addToScope().

When an expression for one variable refers to another variable, then the value of the first variable obviously depends on the value of the second. If the value of the second is modified, then it is important that the value of the first reflects the change. This dependency is automatically handled. When you call getToken(), the expression will be reevaluated if any of the referenced variables have changed values since the last evaluation.

3.6 Expressions

Ptolemy II includes a extensible expression language. This language permits operations on tokens to be specified in a scripting fashion, without requiring compilation of Java code. The language was designed to be extremely succinct, using overloaded operators instead of verbose references to methods in the token classes.[1] The expression language can be used to define parameters in terms of other parameters, for example. It is also used to provide end-users with the ability to describe simple stateless actors without resorting to writing Java code through the Expression actor. The expression language is also used to give guards and resets for finite state machines in an intuitive fashion. The use of the expression language is described in volume 1.

The expression language is extensible. The extension mechanism is based on the reflection package in Java used to add primitive functions and constants to the expression language. The expression language is also purely functional, meaning that it lacks sequencing constructs and side effects. Building state and sequencing into models is done through the use of models of computation, allowing a much richer set of concurrent control structures than is possible with traditional imperative languages. The language is higher-order, since it is integrated with the the FunctionToken class. This allows for

1. The Ptolemy II expression language uses operator overloading, unlike Java. Although we fully agree that the designers of Java made a good decision in omitting operator overloading, our expression language is used in situations where compactness of expressions is extremely important. Expressions often appear in crowded dialog boxes in the user interface, so we cannot afford the luxury of replacing operators with method calls. It is more compact to say “2*(PI + 2i)” rather than “2.multiply(PI.add(2i)),” although both will work in the expression language.

new functions to be easily declared as part of a model, using expressions and for these expressions to be manipulated and passed through a model as data. Because the expression language is side-effect free this mechanism does not interact in unexpected ways with concurrent models of computation. Lastly, the expression language is strongly typed, allowing transparent integration with the static type checking of components specified using expression. When combined with the higher-order constructs the resulting language has the feel of typed lambda calculus.

3.7 Unit System

The unit system in Ptolemy II is based on the paper “Automatic Units Tracking” by Christopher Rettig [122]. The basic idea is to define a suite of parameters to represent the various measurement units of a unit system, such as “meter,” “cm,” “feet,” “miles,” “seconds,” “hours,” and “days.” In each unit category (“length” or “time” for example), there is a base unit with respect to which all the others are specified. If the base unit of length is meters, then “cm” (centimeter) will be specified as “0.01 * meters”. Derived units are specified by just multiplying and dividing base units. For example “newton” is specified as “meter * kilogram / second^2”.

The unit parameters contain tokens just like other parameters. To track units, the category information is stored together with measurement data in scalar tokens, and is used when arithmetic operations, such as add() and multiply(), are performed. The subclasses of ScalarToken, including IntToken and DoubleToken, override these methods to perform unit checking.

The ptolemy.data.unit package provides three classes (BaseUnit, UnitCategory, and UnitSystem) that allow a unit system to be specified using MoML, as illustrated in figure 3.6. When such a unit system is added to the model shown in figure 3.7, the units can be used in expressions to specify the value of actor parameters. The displayed result of executing the model is “10.0 * m / s”.

Several basic unit systems are provided with Ptolemy II. In the Vergil graph editor, they appear in the utilities library. A unit system added to a composite actor can only be used inside that actor. The

```
<property name="Sample" class="ptolemy.data.unit.UnitSystem">
  <property name="m" class="ptolemy.data.unit.BaseUnit" value="1.0">
    <property name="Length" class="ptolemy.data.unit.UnitCategory"/>
  </property>
  <property name="cm" class="ptolemy.data.expr.Parameter" value="0.01*m"/>
  <property name="s" class="ptolemy.data.unit.BaseUnit" value="1.0">
    <property name="Time" class="ptolemy.data.unit.UnitCategory"/>
  </property>
  <property name="ms" class="ptolemy.data.expr.Parameter" value="0.001*s"/>
</property>
```

FIGURE 3.6. A sample unit system.

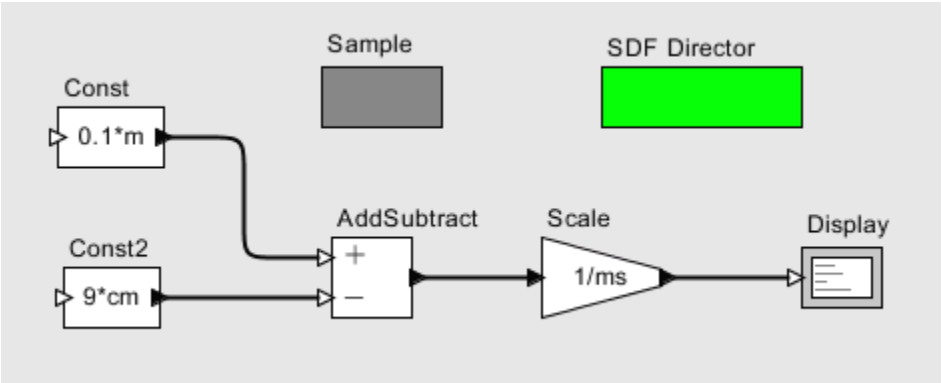


FIGURE 3.7. A model that uses the sample unit system.

user can customize a unit system by adding units, or create new unit systems based on those provided.

The current implementation of unit systems has the following limitations:

- Only scalar values can have units.
- The result of calling a function on a value with units is unit-less.

3.8 The Static Unit System

This section presents the static units system in Ptolemy. In contrast to the unit system in the previous section, which is dynamic, the purpose of the static unit system is to analyze a model before it is run. In particular, the static units system is used to analyze the structure of the model and determine if it is correct in terms of the units of measure.

In the dynamic unit systems units of measure is an integral part of a data value, i.e. it is encapsulated as part of the data Token. In contrast, in the static unit system information about units of measure is in the form of specifications attached to ports. This information can be interpreted as a implying that any data Token that passes through the port at run-time will have those unit specifications. That is, it is a constraint that must be met when the model is run. If the static unit system can determine that constraints of a model will be met at run-time then the model is said to be *units consistent*.

As an example, consider part of the model that is used in the StaticUnits demonstration and is

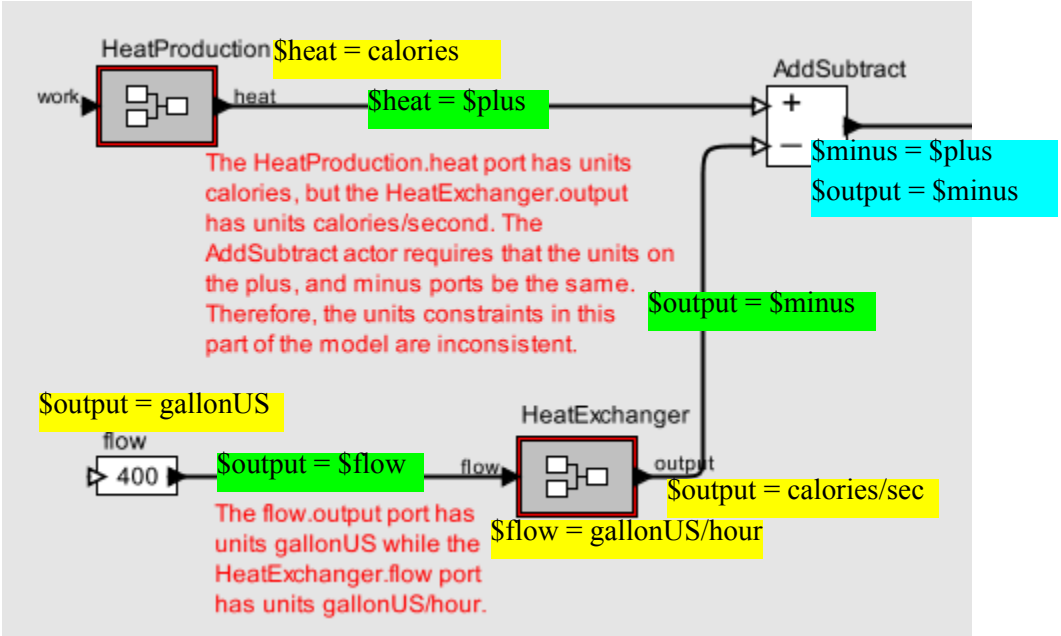


FIGURE 3.8. Part of the StaticUnits demonstration. The colored boxes indicate the unit constraints in this part of the model.

shown in Figure 3.8. There are several unit constraints shown here. Each constraint is in the form of an equation where variables begin with a "$" and refer to ports in the model. The constraints with green background specify that a port will pass data with specific units. For example, the equation "$heat = calories" next to the HeatProduction actor indicates that any data passing through the heat port will be in units of calories. The constraints with cyan background specifies the relationship among ports on a

particular actor. In this case the AddSubtract actor requires that the plus, minus, and out ports all have the same units without specifying what those units will be.The constraints with the yellow background exist when ports on different actors are connected via a relation.

It can be seen that equations of the model in Figure 3.8 are inconsistent. For example, the equations

```
$output = gallonUS
$output = $flow
$flow = gallonUS/hour
```

can not all be true.

In a real sense, static unit specifications are an extension of conventional data types. For example, a datum may be of type double, but, in addition, also be known to represent calories/sec. It is tempting to extend the analytic techniques applicable for conventional data types to the realm of units of measure. However, conventional data types analytic techniques do not appear to be effective in dealing with static unit specifications. Lattices defined on data type inequalities is the basis for powerful techniques for analyzing data types. Although type lattices can be defined for units specifications they seem trivial, and have not lead to any useful techniques. In contrast, the relationship amongst the units specifications of a model are best characterized with a set of equations. This approach is the basis for determining if a model is units consistent.

A strategic goal in the design and implementation was, both, to leave the dynamic unit capability intact, and to re-use the dynamic unit components where possible.

3.8.1 Unit Systems

A unit system is based on 1) an ordered set $\{D_1, D_2 \ldots, D_n\}$ where each $\{D_i\}$ represents a dimension, and 2) a base unit for each dimension. The intent is that each dimension is orthogonal to all other dimensions, and that any unit of measure can be expressed as a combination of the base units. An example of a unit system in widespread use is the International System of Units shown in Table 8

Index	Dimension	Base Unit
1	Length	Meter
2	Time	Second
3	Temperature	Kelvin
4	Mass	Kilogram
5	Current	Ampere
6	Substance	Mole
7	Luminosity	Candela

Table 8: System International Unit System

In order to simplify the presentation of examples in this section the Simple unit system shown in

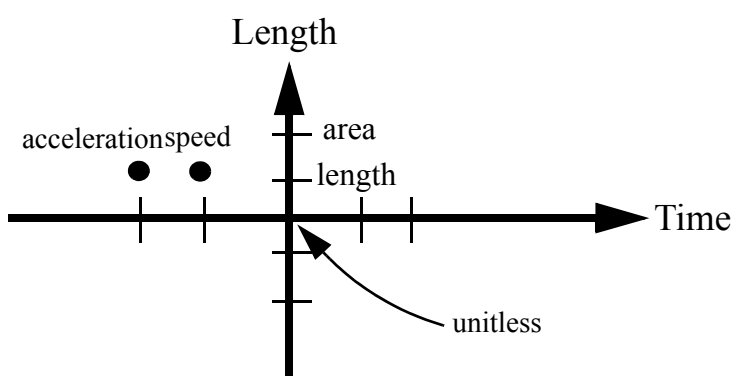
Index	Dimension	Base Unit
1	Length	Meter
2	Time	Second
3	Mass	Kilogram

Table 9: Simple Unit System

Table 9 will be used throughout.

3.8.2 Units of Measurement Algebra

The *type* of a unit is expressed as $\langle e_1,\ldots,e_n\rangle$ where each e_i represents the exponent of the corresponding dimension. For example, $\langle 1, -1, 0\rangle$ (i.e. Length/Time) is a type in the Simple unit system, and is commonly referred to as speed. The set of types for a unit system are a set of points in N-dimensional space. For example, a unit system with just the categories Length and Time (i.e. without Mass) would look like



A type is said to be *singular* if it has the property that just one of the exponents is 1 and the rest are 0. The type $\langle 0, \ldots, 0\rangle$ is said to be the *unitless* type.

A *unit* is obtained by combining a scale with a type. A unit U is expressed as

$$U = \alpha\langle e_1,\ldots,e_n\rangle \qquad \text{where} \qquad \alpha \in \text{Reals} \tag{1}$$

where α is the scale of U. Please note that (1) does not imply the multiplication of $\langle e_1,\ldots,e_n\rangle$ by α. In particular, $\alpha\langle e_1,\ldots,e_n\rangle \neq \langle \alpha e_1,\ldots,\alpha e_n\rangle$ A unit is said to be *singular* if its type is singular. A unit is said to be *basic* if $\alpha = 1.0$.

Refer to Table 10 for some examples of units.

Unit	Descriptive Form(s)	Scale	Type	Properties
1.0<1, 0, 0>	Meter, m	1.0	<1, 0, 0>	basic, singular
1.0<0, 1, 0>	Sec, s	1.0	<0, 1,0>	basic, singular
1.0<0, 0, 1>	Kilogram, kg	1.0	<0, 0, 1>	basic, singular
0.01<1, 0, 0>	Centimeter, cm	0.01	<1, 0, 0>	singular
3600<0, 1, 0>	Hour, hr	3600	<0, 1, 0>	singular
0.453592<0, 0, 1>	Pound, lb	0.453592	<0, 0, 1>	singular
1.0<1, -1, 0>	meters/sec, m/s	1.0	<1, -1, 0>	basic
0.44704<1, -1, 0>	miles/hour, mph	0.44704	<1, -1, 0>	
9.80665<1, -2, 0>	g (gravity)	9.80665	<1, -2, 0>	
3.14158<0, 0, 0>	pi, π	3.14159	<0, 0, 0>	unitless

Table 10: Examples of Units

The unit *multiplication* of two units is the vector addition of the two types and the "normal" multiplication of the two scales. That is

$$U_1 \times U_2 = a_1 * a_2 \langle e_{1,1} + e_{1,2}, e_{2,1} + e_{2,2} \ldots e_{n,1} + e_{n,2} \rangle \text{ where } \left\{ \begin{array}{l} U_1 = a_1 \langle e_{1,1}, e_{2,1}, \ldots, e_{n,1} \rangle \\ U_2 = a_2 \langle e_{1,2}, e_{2,2}, \ldots, e_{n,2} \rangle \end{array} \right\}$$

Unit *exponentiation* follows from multiplication, i.e.

$$U^x = a^x \langle xe_1, xe_2, \ldots, xe_n \rangle$$

The notion of a unit variable is assumed and has the properties commonly found in computational systems. A unit *expression* has the form

$$U \times X_1^{P_1} \times \ldots \times X_q^{P_q} \text{ where } \left\{ \begin{array}{l} \text{U is a unit} \\ X_i^{P_i} \text{ is the variable } X_i \text{ raised to the } P_i \text{ power} \end{array} \right\}$$

A unit *equation* has the form

$$U_X \times X_1^{P_1} \times \ldots \times X_q^{P_q} = U_Y \times Y_1^{S_1} \times \ldots \times Y_r^{S_r} \text{ where } \left\{ \begin{array}{l} \text{each } X_i, Y_j \text{ is a variable} \\ U_X \text{ and } U_Y \text{ are Units} \end{array} \right\} \quad (2)$$

An equation is said to be in *canonical* form if the left side consists solely of powers of variables and the right side is a unit, i.e.

$$V_1^{P_1} \times \ldots \times V_N^{P_N} = U$$

Any equation in the form of (2) can be translated to an equivalent canonical form by first multiplying both sides of (2) by $U_X^{-1} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r}$ yielding

$$X_1^{P_1} \times \ldots \times X_q^{P_q} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r} = U_Y \times U_X^{-1} \tag{3}$$

The right side of (3) can be further reduced since U_Y and U_X are both units, i.e. they are not variables. If U is the value of the expression $U_Y \times U_X^{-1}$ then (3) can be rewritten as

$$X_1^{P_1} \times \ldots \times X_q^{P_q} \times Y_1^{-S_1} \times \ldots \times Y_r^{-S_r} = U \tag{4}$$

which has just powers of variables on the left side and a unit on the right side.

3.8.3 Descriptive Form Language

The descriptive form of a unit is a natural language string that humans normally use when referring to a unit of measure. Table 10, above, shows some examples of descriptive forms. In the Static Unit System descriptive forms are realized through a formal language, called Descriptive Form Language, and its associated grammar shown in Figure 3.9

```
Root ::= UnitEquation | UnitExpression
UnitEquation ::= UnitExpr "=" UnitExpr
UnitExpr ::= UnitTerm {"/" UnitTerm | "*" UnitTerm | UnitTerm}*
UnitTerm ::= UnitElement
            |UnitElement "^" <Number>
            |<Number>
            |"(" UnitExpr ")"
UnitElement ::= Unit| Variable
Variable ::= "$" <String>
```

FIGURE 3.9. BNF for Descriptive Form Language

For example, the parse tree for the expression “gallons/second” is shown in figure 3.10

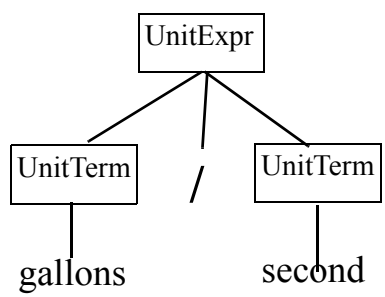


FIGURE 3.10. Parse Tree for “gallons/sec”

The Static Unit System parser, called UParser, is generated using JavaCC which is used to generate the PtParser, the expression parser in Ptolemy. See “Generating the parse tree” on page 4-83. One key difference is that the generation of UParser does not start from a JJTree specification. Instead the DFL grammar is specified in the UParser.jj which is input directly to JavaCC.

If it were possible it would have been advantageous to make DFL a subset of the expression language. However, there are two differences between the DFL and the data expression language that preclude this possibility. First, multiplication in DFL can be expressed via concatenation of the

multiplicands. For example, moment arm can be expressed as “foot pound” in DFL. There is no rule in the data expression grammar that provides for this construction. DFL could be modified so that multiplication requires a “*” operator, as in “foot*pound”(in fact, as a convenience, DFL accepts this construction). However, the form “foot pound” has been in use for decades and it was judged that casual users of Ptolemy would find the requirement of the “*” operator to be awkward.

The second difference stems from the necessity to have variables distinquishable from unit labels.The data expression language does not have a way to explicitly distinguish variables, as it is clear from the context. The example presented in Figure 3.8 shows the descriptive form “$heat = calories” expressing the constraint that the port with name heat has unit calories. Without the “$” to distinguish heat as a port name the parser would try to interpret heat as a unit.

3.8.4 Implementing the Static Unit System in Ptolemy

This subsection presents the internal form that is implemented as a set of classes in Ptolemy. Fig-

ure 3.11 presents the UML static structure for these classes.

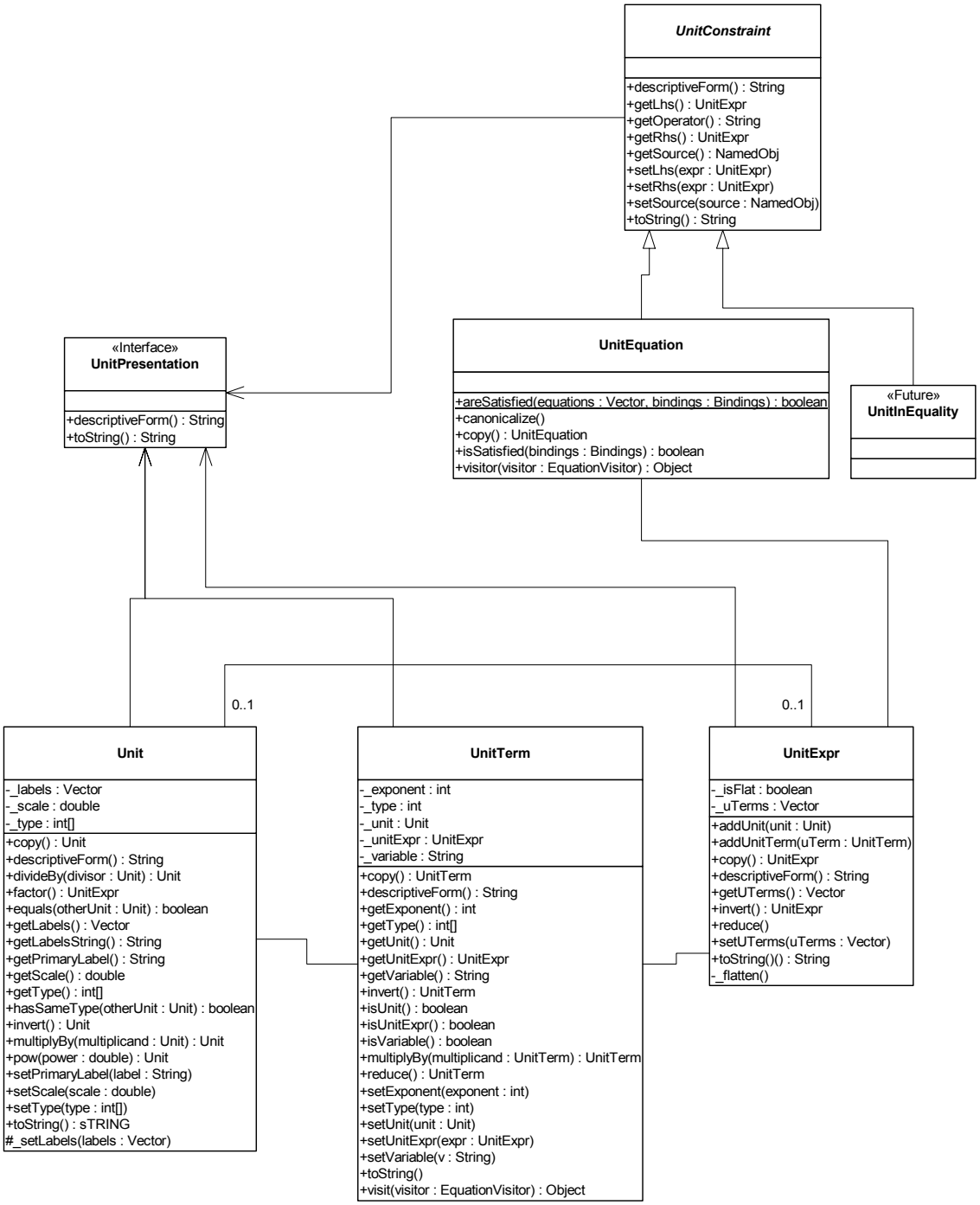


FIGURE 3.11. Static Structure of classes used ti implement units, unit expressions and unit equations.

3.8.5 The Unit Library

The dynamic units system is architected so that a unit system is associated with a model. Further, different models can have different unit systems. In contrast, the static unit systems architecture provides the equivalent functionality with a UnitLibrary. However, there is one common UnitLibrary in the Ptolemy system, and it is used by all models requiring the services of the static unit system. The UnitLibrary is loaded the first time Ptolemy attempts any operations that will require the UnitLibrary be present.

The specifications for a UnitLibrary are contained in a file with the exact same format as is used by the dynamic unit system. (Presently, the name of the file is hardwired to be ptolemy/data/unit/SI.xml.) When loaded by the static unit system a UnitSystem appropriate for the dynamic unit system is created as a side effect. It is the subsequent processing of this UnitSystem that creates the UnitLibrary.

Figures 3.12 and 3.13 show parts of this file

```
 <property name="cm" class="ptolemy.data.unit.BaseUnit" value="1.0">
  <property name="Length" class="ptolemy.data.unit.UnitCategory"/>
  </property>

<property name="second" class="ptolemy.data.unit.BaseUnit" value="1.0">
  <property name="Time" class="ptolemy.data.unit.UnitCategory"/>
  </property>
```

FIGURE 3.12. The Length and Time BasicUnits specifications from the SI.xml file

```
<property name="lightSpeed" class="ptolemy.data.expr.Parameter"
            value="299792458.0*meter/second"/>
<property name="gallonUS" class="ptolemy.data.expr.Parameter"
             value="3785.412*cm^3"/>
<property name="pi" class="ptolemy.data.expr.Parameter" value="3.1415"/>
<property name="planckConstant" class="ptolemy.data.expr.Parameter"
             value="hBar*2*pi"/>
```

FIGURE 3.13. The lightSpeed and gallonUS non-BasicUnit specifications from the SI.xml file.

Although the file format is identical for the two unit systems there is one additional requirement that the static unit system imposes. Non-BasicUnits are specified as a Ptolemy Parameter. In essence the dynamic unit system allows external references to units outside the UnitSystem. For example, the Parameter pi is defined elsewhere in the Ptolemy system and any UnitSystem can refer to that definition. In contrast, the static unit system is based on an architecture where the UnitLibrary is self contained. This is necessary in order to distinguish the case where a Parameter is not a unit. Therefore, for example, pi must be specified in the file as shown in figure 3.13.

3.8.6 Generating Descriptive Forms

In order to present the user with the results of the solver it is necessary to generate the descriptive form from the internal representation of a unit. If the unit exists in the UnitLibrary then generation requires only that the descriptive form be retrieved from the UnitLibrary.

However, it is often the case that the unit does not exist in the UnitLibrary. In the example shown in Figure 3.8 there is an inconsistency between the Flow.output port with units gallonUS and the HeatExchanger.flow port with units gallonUS/hour. The following table has the relevant information

	Descriptive Form	Internal Form	Source
1	gallonUS	3785.412<3,0,0>	UnitLibrary
2	hour	3600<0,1,0>	UnitLibrary
3	gallonUS / hour	1.05<3, -1,0>	derived by parsing the descriptive form
4	to be generated	.000277<0,-1,0>	derived by solving for X in the equation $378.412\langle 3,0,0\rangle \times X = 1.05\langle 3,-1,0\rangle$

in determining the descriptive form of the transformation required to remove this inconsistency.

The internal form in lines 1 and 2 are values obtained from the UnitLibrary. The internal form in line 3 was obtained by parsing the descriptive form "gallonUS/hour" which causes the calculation 3785.412<3,0,0> / 3600<0,10> = 1.05<3,-1,0> to take place. The internal form in line 4 is the result of solving for X in the equation $378.412\langle 3,0,0\rangle \times X = 1.05\langle 3,-1,0\rangle$. That is, X is the transformation required to remove the inconsistency arising from the sending port having units gallonUS to a receiving port having units gallonUS/hour. In order, for this result to be communicated to the user a descriptive form for 0.000277<0,-1,0> must be generated. By noting that 0.000277<0,-1,0> = 1 / 3600<0,-1,0> it can be determined that the descriptive form for 0.000277<0,-1,0> is 1/hour.

Stated formally, generating the descriptive form for a unit U requires that U be factored such that $U = U_1^{P_1} \times \ldots \times U_N^{P_N}$ where each U_i is in the UnitLibrary. In theory, the complexity of this factorization is infinite since the range of each P_i is infinite, and there is no limit on the size of N. By limiting N and the possible values of P_i the complexity can be made at least finite. In the current implementation $N \le 2$ and $P_i \in \{-2,-1,1,2\}$. If a factorization does not exist under these limitations the descriptive form that is "generated" is just a string representation of the unit. I.E., something like "13.27E17<1, -3, 2, 0>". In practice, this seems to be an acceptable limitation.

3.8.7 UnitsConstraint Solver

The purpose of the solver is to determine if the unit constraints of a model are consistent. It does this by transforming the unit constraints to a set of unit equations in canonical form and then performing a modified form of Gaussian elimination.

For a set of M unit equations let $\{V_1,\ldots,V_N\}$ be the set of variables that occur in any of the set of unit equations. The kth unit equation can then be transformed to its canonical form $V_1^{P_{k_1}} \times \ldots \times V_N^{P_{k_N}} = U_k$. The set of equations in canonical form is represented by a data

structure called a powers matrix that is shown in Figure 3.14

$$\begin{array}{ccc|c} V_1 & \cdots & V_N & \\ \hline P_{1,1} & \cdots & P_{1,N} & U_1 \\ \vdots & \ddots & \vdots & \vdots \\ P_{M,1} & \cdots & P_{M,N} & U_M \end{array}$$

FIGURE 3.14. Power Matrix

A power matrix is said to be *reducable* if there exists a row k that has all but one of the $P_{k,j}$ being 0. Eliminate is an operator that can be applied to any reducable power matrix. The result of the eliminate operator is another power matrix. Let $P_{k,l}$ be that element not equal to 0, then the resulting power matrix has the property that all of the $P_{i,l}$ in column l will be 0 except for $P_{k,l}$ which will have the value 1. See Figure 3.15

$$\begin{array}{ccccc|c} V_1 & \cdots & V_l & \cdots & V_N & \\ \hline P_{1,1} & \cdots & P_{1,l} & \cdots & P_{1,N} & U_1 \\ \vdots & & \vdots & & \vdots & \vdots \\ 0 & \cdots & P_{k,l} & \cdots & 0 & U_k \\ \vdots & & \vdots & & \vdots & \vdots \\ P_{M,1} & \cdots & P_{M,l} & \cdots & P_{M,N} & U_M \end{array} \xrightarrow{\text{Eliminate(k,l)}} \begin{array}{ccccc|c} V_1 & \cdots & V_l & \cdots & V_N & \\ \hline P_{1,1} & \cdots & 0 & \cdots & P_{1,N} & U_1' \\ \vdots & & \vdots & & \vdots & \vdots \\ 0 & \cdots & 1 & \cdots & 0 & U_k' \\ \vdots & & \vdots & & \vdots & \vdots \\ P_{M,1} & \cdots & 0 & \cdots & P_{M,N} & U_M' \end{array}$$

FIGURE 3.15. The eliminate operation. The kth row must have all 0 except for the lth column. The result is that the lth column will be all zeros except $P_{k,l} = 1$

The eliminate operation is accomplished in two steps. The first step replaces $P_{k,l}$ with 1, and the second step replaces all of the other $P_{i,l}$ s in the lth column with 0. To accomplish the first step note that the kth row represents the equation $V_k^{P_{k,l}} = U_k$. If both sides of this equation are raised to the $1/P_{k,l}$ this equation becomes $V_k^1 = U_k^{1/P_{k,l}}$. Thus, $P_{k,l}$ has been replaced with, and $U_k' = U_k^{1/P_{k,l}}$

The second step requires that it be assumed that the value of the variable V_l is in fact $U_k^{1/P_{k,l}}$. Any other row i that is not the kth row represents the ith equation $V_1^{P_{i_1}} \times \ldots \times V_l^{P_{i_l}} \times \ldots \times V_N^{P_{i_N}} = U_i$. Multiplying both sides of this equation by $V_l^{-P_{i_l}}$ yields

$$V_1^{P_{i_1}} \times \ldots \times V_l^0 \times \ldots \times V_N^{P_{i_N}} = U_i V_l^{-P_{i_l}}$$
$$= U_i U_k^{-P_{i,l}/P_{k,l}}$$

Thus, $P_{i,l}$ has been replaced with 0, and $U_i' = U_i U_k^{\frac{-P_{i,l}}{P_{k,l}}}$

A power matrix is said to be *inconsistent* if there exists a row k with all $P_{k,j} = 0$ but $U_k \neq I$. Note that if a power matrix M is inconsistent then no elimination on M can yield a power matrix that is consistent. A power matrix is said to be *ambiguous* if there exists a column l with more than 1 of the $P_{i,l}$ not equal to 0. A power matrix that is consistent and non-ambiguous yields a set of bindings for the variables. a power matrix is said to be unique if there does not exist any column l where all of the $P_{i,l}$ are 0.

Gaussian elimination is the repeated application of the eliminate operation that terminates when the power matrix can not be further reduced. The final non-reducable power matrix is then used to determine the status of the original set of unit equations. Let M_r be the power matrix that resulting from a Gaussian elimination. If M_r is inconsistent or ambiguous then the set of unit equations does not have a solution. That is, there is no set of bindings for the variables that will cause all of the unit equations to be satisfied. If M_r is consistent and non-ambiguous then there does exist a set of bindings for the variables that will cause the unit equations to be true. Further, if M_r is unique then there exist a binding set that includes all the variables that will cause the unit equations to be true. Figure 3.16 shows examples of inconsistent and ambiguous power matrices.

V_1	V_2	
0	1	calories
0	0	calories/sec

Inconsistent

V_1	V_2	
1	0	calories
1	0	calories/sec

Ambiguous

FIGURE 3.16. Examples of inconsistent, ambiguous power matrices

Figure 3.17 shows an example of a power matrices that are consistent, and non-ambiguous.

V_1	V_2	
0	1	calories
1	0	calories/sec
0	0	Identity

V_1	V_2	V_3	
0	1	0	calories
1	0	0	calories/sec
0	0	0	Identity

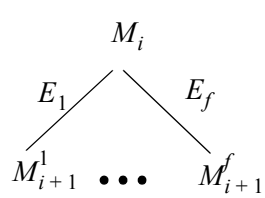
FIGURE 3.17. Consistent, non-ambiguous power matrices. The binding set for the left power matrix is {(V_1, calories), (V_2, calories/sec.)}. The binding set for the right power matrix is{(V_1, calories), (V_2, calories/sec.), (V_3, <unbound>)}

3.8.8 Minimal Span Solutions

Usually an inconsistent set of equations can be made consistent by modifying one or more of the equations and/or changing the membership of the set. In general, however, an inconsistent set of equations does not have a single cause for the inconsistency. That is, it is the set of equations that is inconsistent, not a particular equation in the set that is inconsistent. Stated another way, the cause of the inconsistency is ambiguous. As a result the algorithm presented in the previous subsection can not provide information about why a set of equations is inconsistent. However, it may be possible to provide information that will help the user determine which modifications are appropriate. This is done by presenting the user with a set of minimal span solutions.

A minimal span solution is a solution for a subset of the model, i.e. a subset of the rows in the powers matrix have been eliminated. Furthermore, the subset is one in which the components are connected. The minimal span solution is inconsistent. Finally, if any component is removed the remaining components are consistent. Stated differently, a minimal span solution is inconsistent, but just barely.

The set of minimal span solutions are derived by an adaptation of the Gaussian elimination algorithm that generates the full solution. A minimal span solution is also obtained by applying a sequence of eliminate operations but terminates when a power matrix M_t is generated that is either inconsistent, or is not reducable. Recall that, in general, a power matrix can have more than one eliminate operation applied to it. Thus, the generation of all possible minimal span solutions for a power matrix M_0 forms a tree with M_0 being the root and each leaf being a power matrix that satisfies the termination condition. The structure of each non-leaf node is



where $\{E_1,\ldots,E_f\}$ is the set of all possible eliminate operations that can be applied to M_i.

As an example, the power matrix for the model in Figure 3.8 is shown in Figure 3.18.

V_1	HeatProdcution.heat
V_2	AddSubtract.plus
V_3	AddSubtract.minus
V_4	HeatExchanger.output
V_5	HeatExchanger.flow
V_6	flow.output

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
-1	1	0	0	0	0	Identity	connect HeatProduction.heat to AddSubtract.plus
0	-1	1	0	0	0	Identity	AddSubtract requires plus = minus
0	0	-1	1	0	0	Identity	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

FIGURE 3.18. Power matrix for model shown in Figure 3.7

One minimum span solution is shown by

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
-1	1	0	0	0	0	Identity	connect HeatProduction.heat to AddSubtract.plus
0	-1	1	0	0	0	Identity	AddSubtract requires plus = minus
0	0	-1	1	0	0	Identity	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-0	0	1/hour	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Inconsistent

Here, the minimal span solution has not been greyed out. The row labelled inconsistent caused the

application of the eliminate sequence to terminate. Note, that there are other rows that could be eliminated, but when this row was produced the sequence terminated because minimal span solutions are being generated. Note, also that the unit shown in the inconsistent row is 1/hour and that if the source of this row where replaced with a two port actor that transformed its input by 1/hour then this inconsistency would be removed.

There are two other minimal span solutions of interest

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
0	0	0	0	0	0	1/sec	connect HeatProduction.heat to AddSubtract.plus
0	1	0	0	0	0	calories/sec	AddSubtract requires plus = minus
0	0	1	0	0	0	calories/sec	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Inconsistent

and

V_1	V_2	V_3	V_4	V_5	V_6		source
1	0	0	0	0	0	calories	assignment of calories to HeatProduction.heat
0	1	0	0	0	0	calories	connect HeatProduction.heat to AddSubtract.plus
0	0	1	0	0	0	calories	AddSubtract requires plus = minus
0	0	0	0	0	0	sec	connect AddSubtract.minus to HeatExchanger.output
0	0	0	1	0	0	calories/sec	assignment of calories/sec to HeatExchanger.output
0	0	0	0	1	0	gallonUS/hour	assignment of gallon/US to HeatExchanger.flow
0	0	0	0	-1	1	Identity	connect HeatExchanger.flow to flow.output
0	0	0	0	0	1	gallonUS	assignment of gallonUS to flow.output

Inconsistent

These two minimal span solutions are related in that they have eliminated the same rows in the power matrix. The first indicated that the inconsistency could be fixed by applying the 1/sec transformation

on the relation that connects HeatProduction.heat to AddSubtract.plus. While the second indicates that the inconsistency could be removed by applying the second transformation to the relation that connects AddSubtract.minus to HeatExchanger.output.

3.8.9 Implementing the Units Constraint Solver

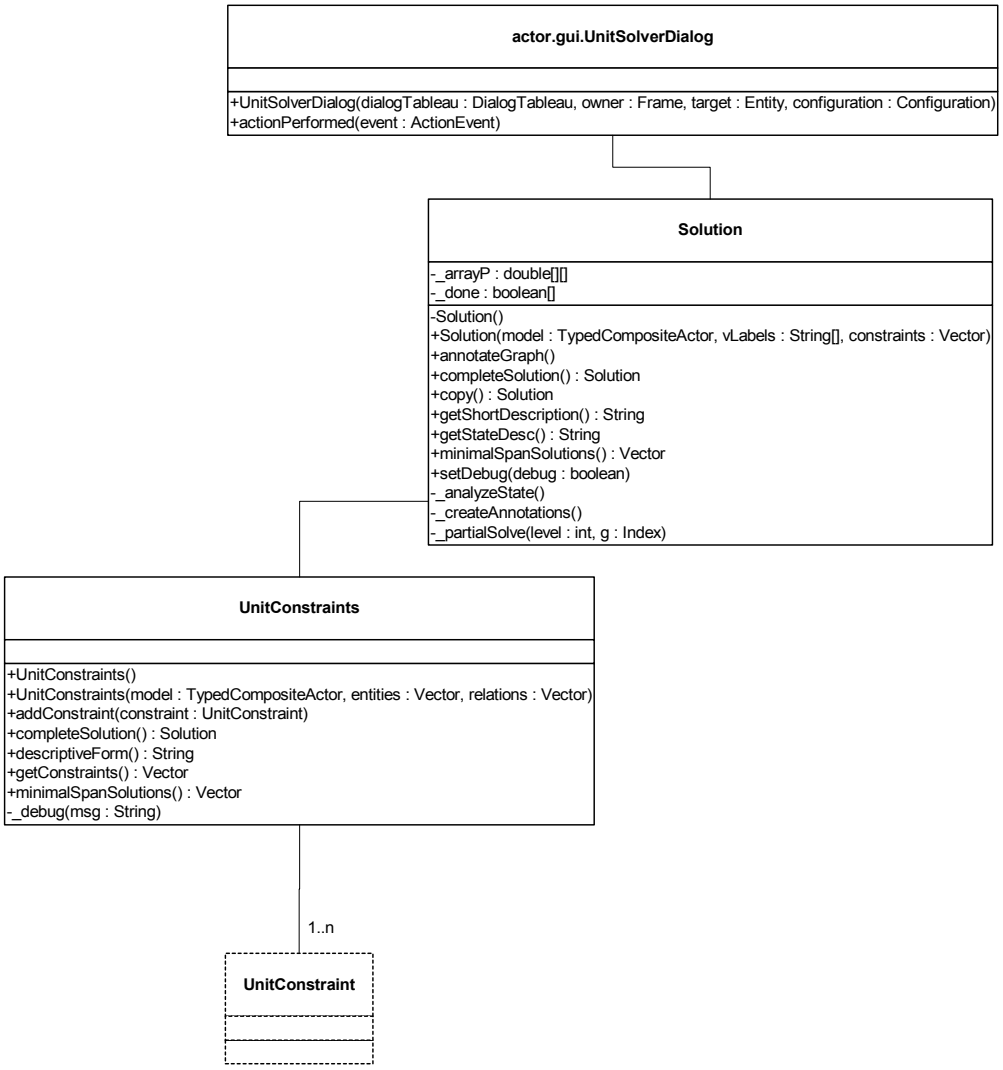


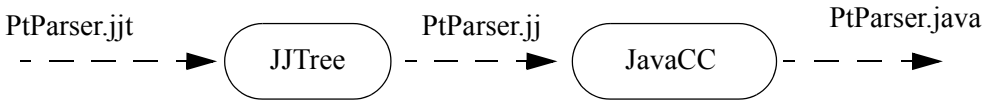
FIGURE 3.19. Static structure of the classes used in the UnitConstraints Solver

Appendix A: Expression Evaluation

The evaluation of an expression is done in two steps. First the expression is parsed to create an *abstract syntax tree* (AST) for the expression. Then the AST is evaluated to obtain the token to be placed in the parameter. In this appendix, "token" refers to instances of the Ptolemy II token classes, as opposed to lexical tokens generated when an expression is parsed.

A.1 Generating the parse tree

In Ptolemy II the expression parser, called *PtParser*, is generated using JavaCC and JJTree. JavaCC is a compiler-compiler that takes as input a file containing both the definitions of the lexical tokens that the parser matches and the production rules used for generating the parse tree for an expression. The production rules are specified in *Backus normal form* (BNF). JJTree is a preprocessor for JavaCC that enables it to create a parse tree. The parser definition is stored in the file PtParser.jjt, and the generated file is PtParser.java. Thus the procedure is



Note that JavaCC generates top-down parsers, or LL(k) in parser terminology. This is different from yacc (or bison) which generates bottom-up parsers, or more formally LALR(1). The JavaCC file also differs from yacc in that it contains both the lexical analyzer and the grammar rules in the same file.

The input expression string is first converted into lexical tokens, which the parser then tries to match using the production rules for the grammar. Each time the parser matches a production rule it creates a node object and places it in the abstract syntax tree. The type of node object created depends on the production rule used to match that part of the expression. For example, when the parser comes upon a multiplication in the expression, it creates an ASTPtProductNode. If the parse is successful, it returns the root node of the parse tree for the given string.

In order to reduce the size of the parse tree, nodes that representing many basic operations are designed to have more than two children, even for binary operations. For instance, the parse tree for the expression "2 + 3 + "hello"" only has one sum node. The children are evaluated in the correct order for the associativity of the operator. In this case, the expression evaluates to the string token with value "5hello".

Note that although functions and constants are registered with the parser, the parser does not actually resolve the values of identifiers. This resolution is performed when the parse tree is actually evaluated. The evaluation process only resorts to registered functions and constants if there are no identifiers defined in the model. This prevents registered functions and constants from unexpectedly shadowing parameters in the model, leading to unexpected behavior. It also allows new functions and constants to be registered without changing the behavior of existing models. Essentially, functions and constants registered with the parser act as a global scope in which all models exist with their own local scopes.

One of the key properties of the expression language is the ability to refer to other parameters by name. Since an expression that refers to other parameters may need to be evaluated several times

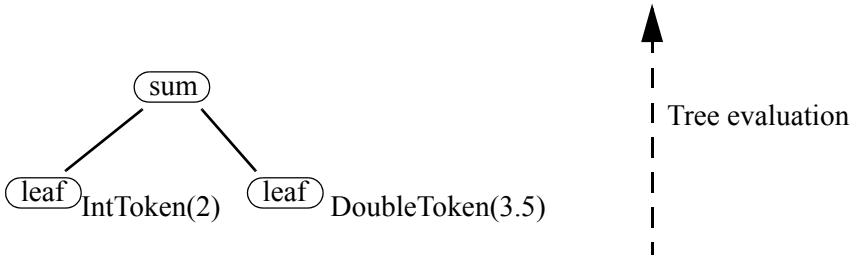
(when the referred parameter changes), it is important that the parse tree does not need to be recreated every time. The classes for representing the parse tree are designed to carry little state, other than the representation of an expression. Generally speaking, users of parse trees, such as the Variable class, cache parse trees for re-evaluation. A new parse tree is only generated when the expression changes. Note, however that the Parser itself is not cached, since it contains a significant amount of internal state.

A.2 Traversing the parse tree

After being generated, the parse tree can be manipulated or traversed, in order to analyze various properties of the original expression. In order to facilitate traversal of the parse tree, the classes representing parse tree nodes implement a visitor design pattern. Each node implements a visit() method that accepts an instance of the ParseTreeVisitor class. When the visit() method of a node is invoked, the node calls an appropriate method of the visitor corresponding to the same node class. The visitor can then operate on the node and recursively invoke the visit method of child nodes to traverse the entire parse tree. This pattern allows the entire logic of a parse tree traversal to be placed in a single class that is largely decoupled from the parse tree itself. Several visitors have been written, and are described below.

A.2.1 Evaluating the parse tree

Parse trees are evaluated using a visitor implemented by the ParseTreeEvaluator class. The parse tree is evaluated in a bottom up manner as each node can only determine its type after the types of all its children have been resolved. As an example consider the input string 2 + 3.5. The parse tree returned from the parser will look like this:



During evaluation, the value of the leaf nodes is first determined, which is trivial in this case, since the values of leaves are constants. These values are then propagated upwards, determining the value of each internal node, until the value of the root node is returned. In this case a DoubleToken with value 5.5 will be returned as the result. If an error occurs during evaluation of the parse tree, an IllegalActionException is thrown with a error message about where the error occurred.

When the ParseTreeEvaluator reaches a instance of the ASTPtLeafNode class that references an identifier, it resolves the identifier into a value through the ParserScope interface. By resolving the values of identifiers through a ParserScope, identifiers can be resolved in different ways depending on how the expression is used. This mechanism is used, for instance to implement the evaluation of function closures and the Expression actors, which interpret expressions differently from parameters. Only if an identifier is not found in scope, is the identifier resolved against the constants registered in the parser.

When the ParseTreeEvaluator reaches a instance of the ASTPtFunctionApplicationNode class, it is handled similarly to a leaf node with an identifier. The name of the function is resolved in the scope, in case the identifier refers to a function closure, an array token, or a matrix token. If the identifier is not found in scope, then reflection to look for that function in the list of classes registered with the parser.

A.2.2 Inferring types of parse trees

The ParseTreeTypeInference class visits parse trees to analyze the type of token resulting from evaluation. For the most part, this operates the same as the ParseTreeEvaluator class, using a ParserScope to resolve the types of identifiers. If identifiers are not present in scope, then they are searched for in the constants or functions registered with the parser.

One difficulty with type inference is that the type of tokens returned from a function invocation can often not be determined from the return type of the Java method. For instance, if the Java method has a return type corresponding to the Token base class, then any token class might be produced. To resolve the types of these methods, the ParseTreeTypeInference class uses Java reflection to find a method with a corresponding name that gives the return type of the original function. For instance, the max() method in the UtilityFunctions class returns the maximum value of an input ArrayToken. Since the ArrayToken can contain any type, the UtilityFunctions class contains a parallel maxReturnType() method that takes a single Type argument, and returns a type. During type inference, this method is found and invoked to properly infer the type returned from the max() method.

A.2.3 Retrieving identifiers in parse trees

The ParseTreeFreeVariableCollector class visits parse trees and extracts the names of all identifiers that need to be resolved to values outside of the expression. In particular, it does not return the names of identifiers that are bound to the arguments of function closures. These identifiers are not accessible outside of the expression. As an example, the expression “foo + bar” has two free variables that must be given values. However, the expression “function(foo:int) foo + bar” has only one free variable, since the identifier “foo” is bound to the argument of the function closure.

A.2.4 Specializing parse trees

The ParseTreeSpecializer class visit parse trees and simplifies them. Primarily, this involves replacing identifier references in leaf nodes with constant values. This operation is an important part of creating FunctionTokens, since the expression inside a function closure can only reference identifiers that are explicitly bound to arguments of the FunctionToken. By specializing the parse tree for the expression, we ensure that the FunctionToken has no dependence on the scope in which it was created. The specializer also analyses the parse tree, finding any internal nodes that are constant after replacing identifiers. These constant nodes are evaluated and replaced by leaf nodes.

A.3 Node types

There are currently fourteen node classes used in creating the syntax tree. For some of these nodes the types of their children are fairly restricted and so type and value resolution is done in the node. For others, the operators that they represent are overloaded, in which case methods in the token classes are called to resolve the node type and value (i.e. the contained token). By type resolution we are referring to the type of the token to be stored in the node.

ASTPtArrayConstructNode. This node is created when an array construction sub-expression is parsed. It contains one child node for each element of the array.

ASTPtAssignmentNode . This node is created when an assignment is parsed. It contains exactly two children. The first child is an ASTPtLeafNode corresponding to the identifier being assigned to and the second child corresponds to the assigned expression.

ASTPtBitwiseNode. This node is created when a bitwise operation (&, |, ^) is parsed. It contains at least two child nodes, and each element has the same operation applied.

ASTPtFunctionApplicationNode. This node is created when a function is invoked. The first child is always a node giving the function that will be invoked. For built-in functions, this child will be a leaf node containing an identifier naming the function. The remaining children correspond to arguments of application from left to right.

ASTPtFunctionDefinitionNode. This node is created when a function definition is parsed. For each argument of the function definition, there are two child nodes. The first child node is a leaf node that contains an identifier for the argument name, while the second gives an expression for the type of the argument. If no type is specified, then a child node is created that evaluates to a type of general. The last child node contains an expression tree that defines the function.

ASTPtFunctionalIfNode. This is created when a functional if is parsed. This node always has three children, the first for the boolean condition and the remaining two children for each branch of the expression.

ASTPtLeafNode. This represents the leaf nodes in the AST. The node contains either a token corresponding to constant values, or a string name for an identifier in the expression. This node contains no children.

ASTPtLogicalNode. This node is created when a logical operation (&&, ||) is parsed. It contains at least two child nodes, and each element has the same operation applied.

ASTPtMatrixConstructNode. This is created when a matrix construction sub-expression is parsed. If the matrix is specified explicitly, then this node contains one child node for each element of the matrix. If the matrix is specified using sequence notation for each row, then the node contains three children for each row of the matrix.

ASTPtMethodCallNode. This is created when a method call is parsed. The first child corresponds to the value the method is being invoked on, while the remaining children correspond to arguments of the method call.

ASTPtProductNode. This is created when an arithmatic product operation (*, /,%) is parsed. It contains at least two child nodes, although the same operation need not be applied to each child. The node contains a list of operations corresponding to the individual operations that need to be applied. This list has one fewer element than the number of children.

ASTPtRecordConstructNode. This is created when a record construct sub-expression is parsed. It contains one node for each value in the record and a list of names corresponding to the label for each value.

ASTPtRelationalNode. This is created when one of the relational operators (!=, ==, >, >=, <, <=) is

parsed. It contains exactly two child nodes.

ASTPtRootNode. Parent class of all the other nodes.

ASTPtSumNode. This is created when a arithmatic summation operation (+, -) is parsed. It contains at least two child nodes, although the same operation need not be applied to each child. The node contains a list of operations corresponding to the individual operations that need to be applied. This list has one fewer element than the number of children.

ASTPtUnaryNode. This is created when a unary negation operator (!, ~, -) is parsed. It always contains exactly one child node.

A.4 Extensibility

The Ptolemy II expression language has been designed to be extensible. The main mechanisms for extending the functionality of the parser is the ability to register new constants with it and new classes containing functions that can be called. However it is also possible to add and invoke methods on tokens, or to even add new rules to the grammar, although both of these options should only be considered in rare situations.

To add a new constant that the parser will recognize, invoke the method registerConstant(String name, Object value) on the parser. This is a static method so whatever constant you add will be visible to all instances of PtParser in the Java virtual machine. The method works by converting, if possible, whatever data the object has to a token and storing it in a hashtable indexed by name. By default, only the constants in java.lang.Math are registered.

To add a new Class to the classes searched for a a function call, invoke the method registerClass(String name) on the parser. This is also a static method so whatever class you add will be searched by all instances of PtParser in the JVM. The name given must be the fully qualified name of the class to be added, for example “java.lang.Math”. The method works by creating and storing the Class object corresponding to the given string. If the class does not exist an exception is thrown. When a function call is parsed, an ASTPtFunctionNode is created. Then when the parse tree is being evaluated, the node obtains a list of the classes it should search for the function and, using reflection, searches the classes until it either finds the desired function or there are no more classes to search. The classes are searched in the same order as they were registered with the parser, so it is better to register those classes that are used frequently first.