A Code Generation Framework for Ptolemy II

by Jeff Tsay
ctsay@cs.berkeley.edu

Technical Memorandum UCB/ERL
Electronics Research Laboratory, Berkeley, CA, 94720 May 19, 2000.

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Committee:

Professor Edward A. Lee
Research Advisor

* * * * * * *

Professor Susan L. Graham
Second Reader

1. Introduction

1.1 Ptolemy II

The Ptolemy II software package [1] provides an environment for simulation of concurrent systems composed of **actors**, which communicate with each other by receiving and sending data through **ports**. Ports are contained within actors, and may be used for input, output, or both. Ports may have a variable number of **channels** that may be connected to channels of other ports. All data sent through ports are encapsulated by **tokens**, of which there are many kinds. The exact semantics of communication are determined by the **domain** in which a system executes. For example, in the Communicating Sequential Processes (CSP) domain, each actor executes in a separate thread, and communication is done through rendezvous (an actor sending data to a port must block until another actor attempts to receive data from the port, and an actor receiving data from a port must block until another actor attempts to send data to the port, at which time the data can be exchanged). Another domain, which is the focus of this project, is the Synchronous Dataflow (SDF) domain, in which all actors may be executed sequentially according to a static schedule. Port I/O requires no blocking and really comes down to reading or writing a buffer.

Actors may also have **parameters**, which may be configured at run-time for more flexibility. Parameters can be queried for and set with token values. Since parameters are public fields of actors, they can be queried and set from outside of the actor class.

An executable system in Ptolemy consists of instances of actors that are contained in an instance of a **composite actor**. A composite actor contains a director, which invokes the following execution methods of actors:

- `preinitialize()`, which is invoked exactly once before simulation begins.
- `initialize()`, which is invoked exactly once after `preinitialize()` is called.
- `prefire()`, which is invoked once per iteration.
- `fire()`, which is invoked several times per iteration, after `prefire()` is called.
- `postfire()`, which is invoked once per iteration, after the last `fire()` is called.
- `wrapup()`, which is invoked exactly once after simulation begins.

Typically, the majority of simulation time is spent invoking `fire()` which does most of the work of an actor. During the invocation of `initialize()`, `prefire()`, `fire()`, and `postfire()`, the actor may read a token from a channel of a port by calling `port.get(ch)`, where `ch` is the channel number. It may write a token to a channel of a port by calling `port.send(ch, t)`, where `ch` is the channel number, and `t` is a token. Finally, it may write to all channels of a port by calling `port.broadcast(t)`, where `t` is a token.

The environment Ptolemy II is rich enough so that systems using the different domains still use the same basic kernel. This richness allows for heterogeneous simulation, in which two or more domains may be used simultaneously in the same

system, and **domain-polymorphic** actors, which are actors that may be used as valid actors in more than one domain. In addition, because data is encapsulated in tokens, actors may also be **type-polymorphic**, meaning that the actor may receive or send more than one kind of token to one of its ports. Token objects are also able to perform elementary operations with other kinds of tokens and convert other kinds of tokens, so actors that do not fully specify the meaning of their operations still may be useful. As an example of polymorphism, consider the AddSubtract actor. The actor executes the following steps in its `fire()` method:

1) Read token(s) from the `plus` port, and "add" them to the initially "zero" result.
2) Read token(s) from the `minus` port, and "subtract" them from the result.
3) Write the result token to the `output` port.

The above sequence of actions is legal in most domains, including SDF, so the actor is domain-polymorphic.

In step 1), instances of Token are read from a port. These tokens may encapsulate integers, double precision floating-point numbers, integers, matrices, or even strings. The addition operation provided by the subclass of Token supplies the real meaning of addition. For tokens that are numbers, arithmetic addition is performed. For tokens that are strings, string concatenation is performed. It is therefore legal to receive different kinds of tokens, making the actor type-polymorphic.

1.2 Motivation for Code Generation for Ptolemy II

While type and domain polymorphism allow for maximum code reuse (an SDF adder actor is not required, nor is an adder that specifically adds matrices of complex numbers), they have the disadvantage of run-time overhead during simulation. A method call to `send()` on a port is undoubtedly slower than directly writing to a buffer. A method call to the `add()` method of a kind of token is undoubtedly slower than directly adding two integers. Herein lies the potential for performance improvements through code transformations of actor source code, if the domain is known and the kinds of tokens can be resolved.

Another reason for code transformation of actor source code is standalone synthesis of code that does not depend (or depends less) on the code in the Ptolemy II software package. The transformed code, while still in the language that the actor was written in, can be converted to C by generic Java to C converters that do not have to understand Ptolemy semantics. This C code could then be compiled and executed efficiently in embedded systems.

2. Related Work

In Ptolemy Classic [2], the predecessor of Ptolemy II, a very different approach to code generation was taken. Code generation was done as a separate domain. Each actor (or **star** as they were called then) was responsible for generating its own code, by supplying additional source code. The source code was allowed to reference special macros. Therefore, the simulation source code and code generated source code were two

different pieces of code. Inherent problems in this approach are additional effort required to use a star in a code generated system and decreased maintainability of each star's source code.

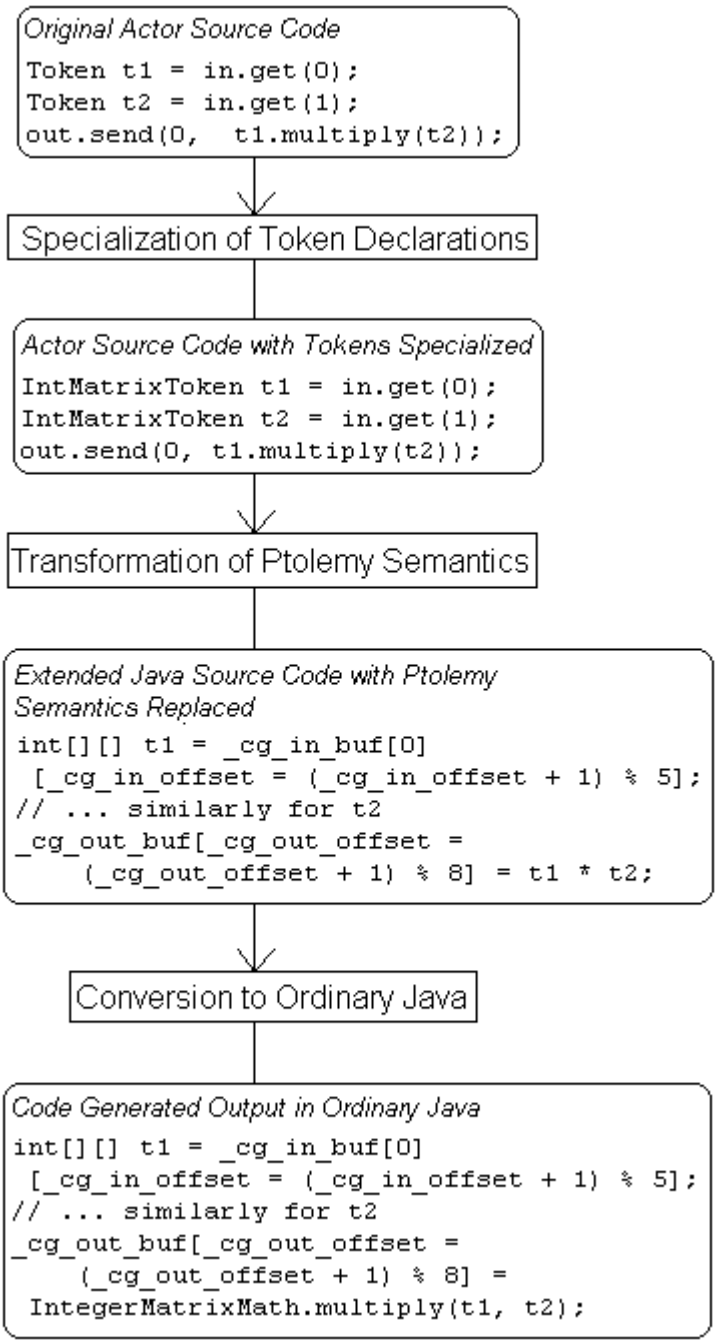
The MathWorks Real-Time Workshop [3] also generates C code from a block diagram specified in Simulink. Simulink, built on top of Matlab, provides a variety of built-in data types including complex numbers, fixed-point numbers, and matrices. Real-Time Workshop can generate code that may be used in single and multi-tasking environments. The code generation process consists of two steps: generation of "C MEX S-functions" from a Simulink block diagram, and generation of C code from these functions. Between these steps, custom C code can be specified by the user to avoid the use of the C MEX S-functions. For each block in the block diagram, the behavior of the code generator can be customized for efficiency. Such behavioral changes are described in another language and compiled by the "Target Language Compiler". By combining automatically generated code based on the existing code used for Simulink blocks, additional code specified by the user, and behavioral changes to the code generator specified by the user, Real-Time Workshop is flexible and able to generate highly optimized code.

3. Code Generation Strategy in Ptolemy II

The goal of this project is to make code generation as painless as possible to users. A user who writes an actor should not need to know how to generate code for it. It should be possible to generate efficient code for polymorphic actors if the configuration of the actor in the system is analyzed. Therefore, the approach taken in Ptolemy II consists of the following steps:

1) The system is configured as usual in preparation for ordinary execution.
2) Each actor is analyzed, in relation to other actors in the system.
3) The source code for each actor is parsed and static semantic analysis is performed, yielding a decorated abstract syntax tree (AST).
4) Declarations of tokens are "specialized", i.e. declarations of instances of abstract tokens are transformed into declarations of instances of concrete tokens. This is done by solving inequalities on the most specific type of token allowable for each declaration.
5) The resulting abstract syntax tree is transformed in a domain-specific way to an intermediate abstract syntax tree, which has extended type rules and conversions. Tokens are replaced with their encapsulated data, and token operations (which are method calls) are replaced with ordinary addition, subtraction, etc. Reads and writes of tokens to port are replaced in a domain-specific way.
6) The AST is reverted back to ordinary Java by adding conversions that widen types and method calls that implement matrix addition, etc.
7) Finally, Java source code is regenerated from the transformed AST.

These steps are shown in the following diagram, which uses as an example a multiplier in SDF:



The code generation process is written entirely in Java.

4. General Compiler Tools

Step 3 of the code generation process requires that compilation be performed on the source code of each actor. The compilation process, however, does not need to generate machine code. Because Ptolemy II actors are written in Java, in particular, code for compilation of Java is required. However, in this project, we attempt to separate code that could be used in any compiler from code that is used only for the Java compiler. It is our hope that the general compiler code would be used in compilers for other languages. The general compiler code is found in the `ptolemy.lang` package.

The Java compiler in Ptolemy II was based on the source code for the Titanium compiler [4]. Titanium is a superset of Java that allows constructs that may be used to optimize programs executed in high-performance environments. The source code for the Titanium compiler itself is written in C++, so we were not able to use the source code without making a few changes because all Ptolemy II packages are written in Java. In addition, a major design change was made. In Titanium, each type of node in the abstract syntax tree is responsible for dealing with itself during each stage of the compilation process. For example, a node representing the addition of two expressions is responsible for figuring out the resulting expression type in the third stage of static resolution, and for generating corresponding Split-C code in the code generation stage. This approach has the following disadvantages:

1) The code to do one coherent operation is spread over all node classes, making the code difficult to maintain and debug. While in C++, the code can be collected in a single file, in Java, it must be placed in the node class definition file.
2) Additional operations can only be added by modifying the source code for each node.

A crucial step of the Ptolemy II code generator is to transform an AST. If the above approach were taken, the source code for each type of node would need to contain general static semantic analysis code as well as code to do operations to do transformations that would only be applicable to analyzing Ptolemy II actors. However, as a general design principle, we want the general Java compiler code to be unpolluted by code for unrelated operations.

We solve this problem with the Visitor pattern [5]. Each node contains only a few access methods, and an `_acceptHere()` method. The `_acceptHere()` method takes as an argument an instance of the interface IVisitor, and passes the node to the appropriate method of the visitor. Each type of IVisitor is responsible for one coherent operation, and has one method for each of the types of nodes that may appear in the AST. Because the nodes have only access methods and the `_acceptHere()` method, the code for each type of node may be generated automatically. Doing automatic generation of subclasses of TreeNode allows the node class hierarchy to be specified in one file, allowing changes in hierarchy to be made more easily.

To illustrate the Visitor pattern more clearly, let us consider a toy language that has only two constructs: a reference to a variable and addition. Assuming both types of nodes extend ExprNode, which extends TreeNode, the node representing addition might look like

```
class PlusNode extends ExprNode {
   public PlusNode(ExprNode expr1, ExprNode expr2) { ... }
   public getExpr1() { return _expr1; }
   public getExpr2() { return _expr2; }

   public void setExpr1(ExprNode expr1) { _expr1 = expr1; }
   public void setExpr2(ExprNode expr2) { _expr2 = expr2; }

   protected Object _acceptHere(IVisitor visitor, LinkedList args) {
       return ((ToyVisitor) visitor).visitPlusNode(this, args);
   }

   protected ExprNode _expr1;
   protected ExprNode _expr2;
}
```

The code for a variable reference might look like

```
class VarNode extends ExprNode {
   public VarNode(String name) { ... }

   public String getName() { return _name; }
   public void setName(String name) { _name = name; }

   protected Object _acceptHere(IVisitor visitor, LinkedList args) {
       return ((ToyVisitor) visitor).visitVarNode(this, args);
   }

   protected String _name;
}
```

The visitor base class for the language would look like

```
class ToyVisitor extends IVisitor {
    public ToyVisitor() {}

    public int traversalMethod() { return TM_CUSTOM; }

    public Object visitPlusNode(PlusNode node, LinkedList args) {
        return null;
    }

    public Object visitVarNode(VarNode node, LinkedList args) {
        return null;
    }
}
```

Visitors in the toy language would then extend ToyVisitor to do some operation on the AST. When the `accept()` method of a node is called, it may call `accept()` on the nodes in its child list either before or after it calls `_acceptHere()` on itself. The `_acceptHere()` method then calls the appropriate method of the visitor.

The disadvantage of the Visitor pattern is that operation code is no longer implicitly inherited if one node class extends another. When such behavior is desired, code reuse can be achieved by manually calling a method that handles an abstract tree node in each visitation method of a node that extends the abstract tree node class.

Based on the Titanium compiler source code, we have converted the following classes for use in compilers:

- **Decl** encapsulates a declaration of some kind. Each declaration has a name and a category. The Decl class extends the class TrackedPropertyMap. Instances of Decl are intended to be unique; for every actual declaration there is exactly one instance of Decl. When referring back to a declaration, a pointer reference to the original instance of Decl must be used instead of copies.
- **Environ** encapsulates an environment containing declarations. Each environment may recursively contain other environments. Methods are provided for searching for declarations non-recursively (i.e. **properly**) and recursively in an environment. These methods return instances of Decl or EnvironIter.
- **EnvironIter** represents an iterator over declarations that match some criteria for their names and categories. When an EnvironIter is created, no attempt is made to find all matches within an environment. Instead, the matching process is done lazily as the iterator is advanced. For convenience, EnvironIter implements java.util.Iterator.
- **TreeNode** represents a node in an abstract syntax tree. All node classes that appear in an abstract syntax tree should eventually derive from TreeNode. Each node contains one **child list**. A child list is a list whose members may be instances of TreeNode or other child lists. The child list is implemented by ArrayList, which provides constant time access to list members. Therefore, in subclasses of TreeNode, it is not necessary to declare explicit fields for child nodes or lists of nodes. TreeNode accepts an instance of IVisitor, and invokes the corresponding method in the visitor class. The `accept()` method also performs automatic visitation of the child list if desired, and eventually calls `_acceptHere()`. In addition, TreeNode provides a `toString()` method that uses reflection to return a meaningful String representation of any subclass of TreeNode. Finally, the `clone()` method is overloaded so that a deep copy of TreeNode is made (nodes or lists in the child list are also cloned). The reason that a deep copy is necessary is that each instance of TreeNode in the AST should be unique and contained by at most one TreeNode, except for special singleton nodes. TreeNode extends the class TrackedPropertyMap, and implements the class ITreeNode.

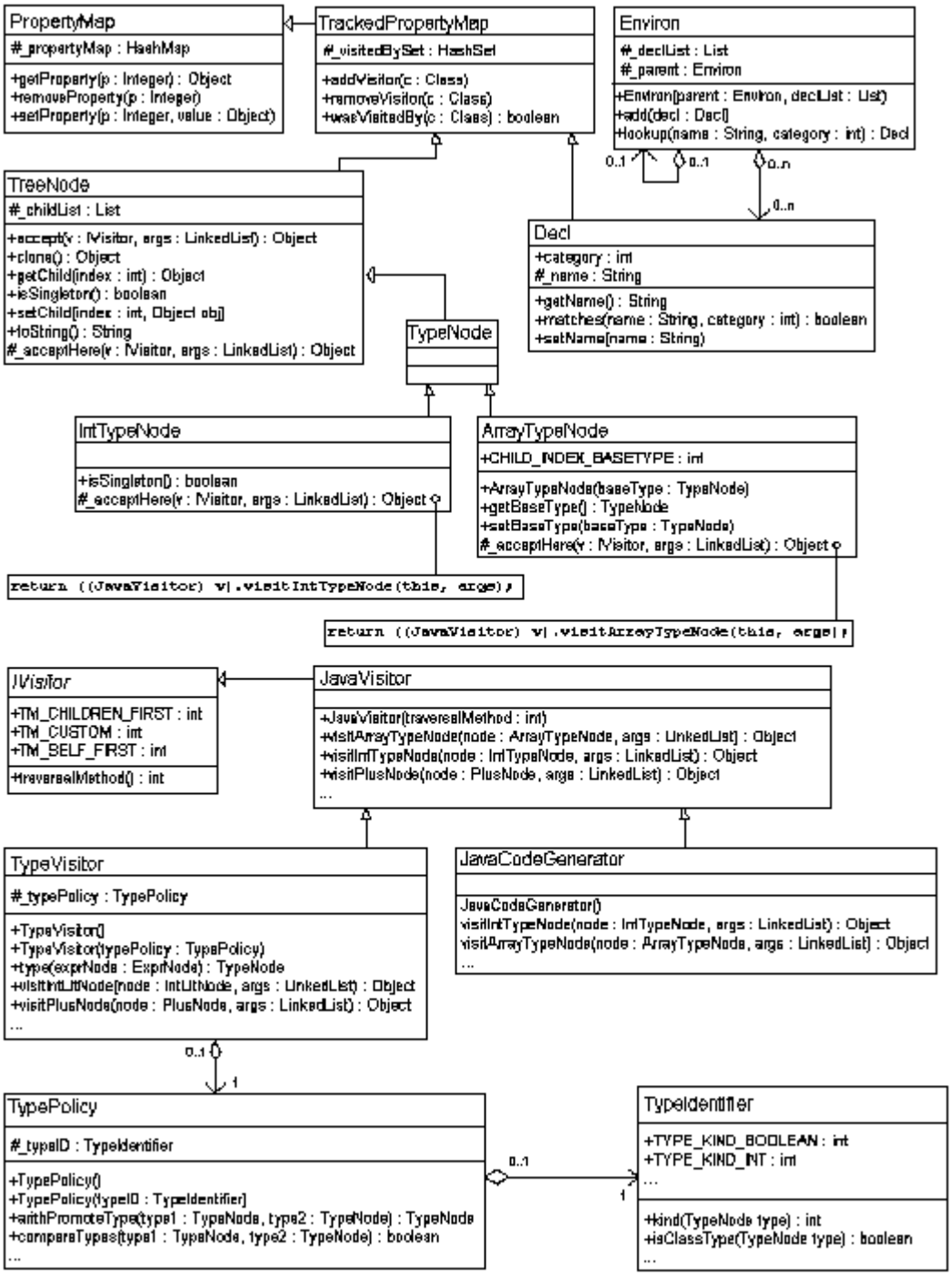
In addition to the code converted from the Titanium compiler, the following classes are also part of the `ptolemy.lang` package:

- **ApplicationUtility** contains methods useful for reporting errors, warnings, trace messages and doing assertions for command-line applications such as compilers. The action taken when encountering an error, warning, or assertion failure can be chosen.
- **PropertyMap** allows properties to be put and retrieved. A property is specified by an instance of Integer, and property values may be any user type. Properties are typically

used to store the result of the operation of a single visitor. Later, a property may be retrieved by another visitor.

- **TrackedPropertyMap** keeps track of which visitors have "visited" the object. By visitation, we mean that either the object is a node in the AST and it has accepted a given type of visitor, or the visitor has in some way dealt with the object already. Visitation is manually marked by calling `addVisitor()` with the Class object associated with a specific type of visitor. TrackedPropertyMap extends the class PropertyMap.
- **TNLManip** provides group of methods that provides convenience methods for constructing and visiting child lists.
- **ITreeNode** is an interface that all TreeNodes must implement. This interface is provided so that interfaces that extend it may be used just as TreeNodes. For example, the interface StatementNode in the Java compiler extends ITreeNode, and therefore instances of StatementNode can also accept visitors (all nodes eventually derive from TreeNode anyway).
- **IVisitor** is an interface that node visitors should implement. TreeNodes accept visitors of type IVisitor, instead of concrete visitor classes that have methods that depend on the language being compiled.
- **ObjectInterrogator** is a static class that contains an `interrogate()` method which uses reflection to determine fields, method return values, and properties of objects, during ordinary execution of a program. A user inputs what he/she wants to inspect, and the resulting value is displayed on the screen. The resulting value is then recursively interrogated. This class can be useful for the "just-in-time debugging" of compilers, if calls to `interrogate()` are made at strategic places in the code to be debugged. `jdb` provides similar capabilities (without support for properties), but executes the entire program slowly.
- **GenerateVisitor** is a standalone program that reads a node class definition file, generates the source code for the nodes and a base class for visitors of the nodes. This class is further described in Appendix A.

The following UML diagram shows the relationships between the key classes:



In this diagram, only two concrete subclasses of TreeNode are shown for simplicity. The diagram also shows classes used in the Java compiler, described next.

5. Java Compiler Front End

This section is heavily influenced by the work in Titanium compiler, as most of the static semantic analysis code was just a conversion of the Titanium compiler source code to Java.

The Java compiler is found in the sub-package `ptolemy.lang.java`. Nodes and the concrete visitor class for a Java AST, JavaVisitor, were generated by GenerateVisitor, and placed in the sub-package `ptolemy.lang.java.nodetypes`. The descriptions of all node types are found in Appendix B.

A subclass of Decl, JavaDecl provides functionality common to declarations in Java, so declarations found as properties of nodes all derive from JavaDecl. The following are the types of declarations in Java:

Description	Class name	Allowed categories	Container	Modifiers	Environ	Source
Package	PackageDecl	CG_PACKAGE	X		X	
Class or Interface	ClassDecl	CG_CLASS, CG_INTERFACE	X	X	X	X
Method or constructor	MethodDecl	CG_METHOD, CG_CONSTRUCTOR	X	X		X
Class field	FieldDecl	CG_FIELD	X	X		X
Local variable	LocalVarDecl	CG_LOCALVAR		X		X
Parameter	FormalParameterDecl	CG_FORMAL		X		X
Statement label	StmtLabelDecl	CG_STMTLABEL				X

If present, the container, modifiers, environment, or source tree node associated with a declaration can be queried and set using methods like `getContainer()` and `setContainer()` which are present in JavaDecl. If a subclass of a declaration does not have a given attribute, an exception will be thrown when `getContainer()` is called on a local variable declaration, for example. In addition, a JavaDecl can be queried to see if it has an attribute by calling a method like `hasContainer()`.

5.1 Parser and Lexical Analyzer

As mentioned previously, the source code for the Java compiler was taken from the Titanium project, with some modifications. Because we intended to write the Java compiler in Ptolemy II entirely in Java, the lexical analyzer and the parser also had to be generated in Java. However, the tools to generate the lexical analyzer and parser in Titanium, `flex` [6] and `bison` [7], respectively, only produce C++ code. Therefore, JLex [8] was chosen as the lexical analyzer generator, and BYACC/J [9] was chosen as the parser generator because they generate Java code. BYACC/J was also chosen because the grammar file is of the same format as `yacc`, with which `bison` is compatible. We were able to create a valid Java grammar definition by making some changes to the `bison` grammar file for Titanium. Titanium-specific rules were deleted, and additional rules for Java 1.2 were added (new uses for modifiers, inner classes, anonymous classes and arrays, `.class` access, and the `strictfp` keyword).

JavaCC [10] and ANTLR [11] were also considered for parser generators but were rejected because the format of each respective grammar file is different. In addition, JavaCC does not handle LALR(1) rules, which the Java Language Specification [12] uses to describe the grammar. The quality and features of ANTLR were found to be high, and it has the advantages of being written in Java, with the source code freely distributed, and having a built-in lexical analyzer.

Nevertheless, BYACC/J was chosen because the time required to convert a `bison` grammar file to an ANTLR grammar file was estimated to be too long. BYACC/J was written in C, and two major modifications to the source code had to be made so that the generated parsers would work correctly.

The first change was that parser tables had to be stored in external files instead of inlined in the source code. The reason is that Java class files have a size limitation of 64 kilobytes, which is acceptable for parsers of trivial languages, but not for a Java parser. The modified BYACC/J writes to external files when generating the parser, and the modified parser code reads from these external files. The problem with this approach is that the parser does not know where to look for the external files, except for the directory in which the parser class file resides. Consequently, any use of the Java compiler must be initiated from the `ptolemy/lang/java` directory.

The second change to the BYACC/J source code was that parser values had to be cloned so that each instance of parser return value is only used once. It seems unbelievable that this bug could exist in BYACC/J, but nevertheless making this change was necessary so that the parser would work correctly. Both of these changes have been mentioned to the author of BYACC/J.

BYACC/J does not have a built-in lexical analyzer, so JLex was chosen to generate the lexical analyzer. JLex takes as input a file very similar in syntax to definition files for `flex`. Since the Titanium source code includes such a file, it was relatively trivial to convert it into an input file for JLex by removing Titanium-specific lexemes and making minor syntactic changes.

Once a source file is parsed, the AST can be displayed on the screen by running the standalone class PrintTree. An example of the textual representation of an AST is shown below:

```
CompileUnitNode
 DefTypes: list
  ClassDeclNode
   Interfaces: <empty list>
   Members: list
    ConstructorDeclNode
     Modifiers: 1
     Name: NameNode
            Ident: OneFM
            Qualifier: AbsentTreeNode (leaf)
           END NameNode
     Params: <empty list>
     ThrowsList: <empty list>
     Body: BlockNode
            Stmts: list
             ExprStmtNode
              Expr: AssignNode
                     Expr1: ObjectNode
                             Name: NameNode
                                    Ident: x
                                    Qualifier: AbsentTreeNode (leaf)
                                   END NameNode
                            END ObjectNode
                     Expr2: IntLitNode
                             Literal: 1
                            END IntLitNode
                   END AssignNode
             END ExprStmtNode
            END list
           END BlockNode
     ConstructorCall: SuperConstructorCallNode
                       Args: <empty list>
                      END SuperConstructorCallNode
    END ConstructorDeclNode
    MethodDeclNode
     Modifiers: 1
     Name: NameNode
            Ident: get
            Qualifier: AbsentTreeNode (leaf)
           END NameNode
     Params: <empty list>
     ThrowsList: <empty list>
     Body: BlockNode
            Stmts: list
             ReturnNode
              Expr: ObjectNode
                     Name: NameNode
                            Ident: x
                            Qualifier: AbsentTreeNode (leaf)
                           END NameNode
                    END ObjectNode
             END ReturnNode
            END list
           END BlockNode
     ReturnType: IntTypeNode (leaf)
    END MethodDeclNode
    MethodDeclNode (Shown in following picture)
```

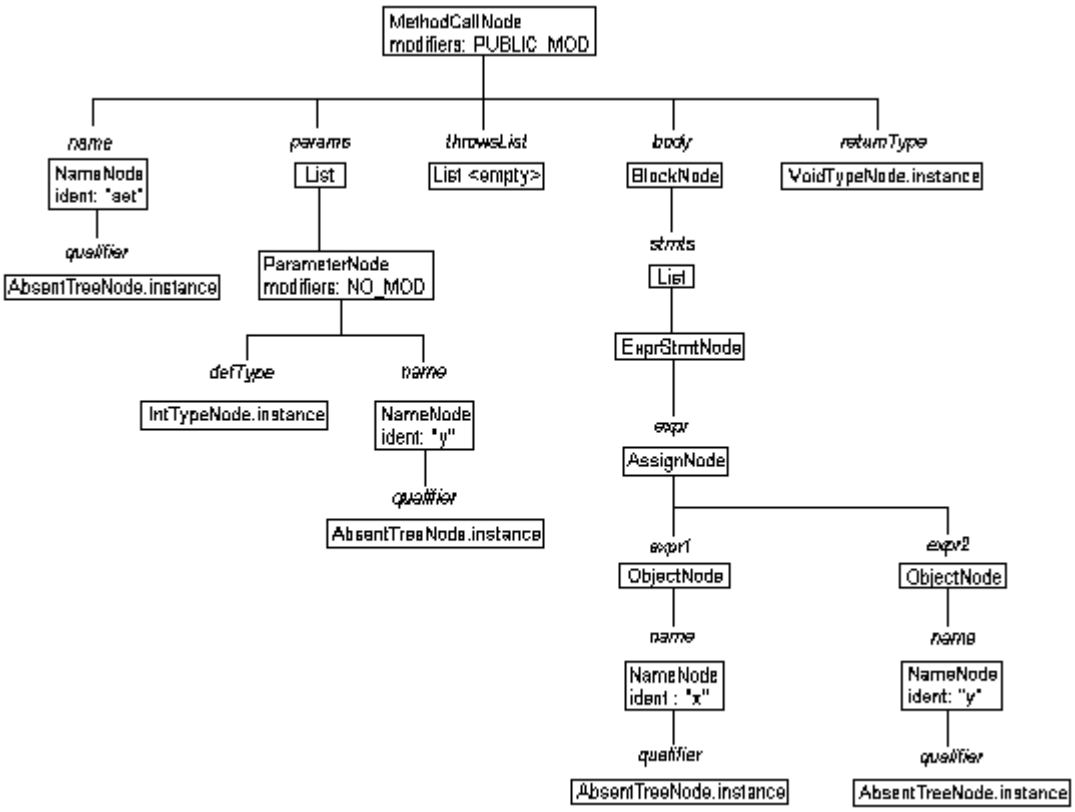
```
    Modifiers: 1
    Name: NameNode
           Ident: set
           Qualifier: AbsentTreeNode (leaf)
          END NameNode
    Params: list
     ParameterNode
      DefType: IntTypeNode (leaf)
      Modifiers: 0
      Name: NameNode
             Ident: y
             Qualifier: AbsentTreeNode (leaf)
            END NameNode
     END ParameterNode
    END list
    ThrowsList: <empty list>
    Body: BlockNode
           Stmts: list
            ExprStmtNode
             Expr: AssignNode
                    Expr1: ObjectNode
                            Name: NameNode
                                   Ident: x
                                   Qualifier: AbsentTreeNode (leaf)
                                  END NameNode
                           END ObjectNode
                    Expr2: ObjectNode
                            Name: NameNode
                                   Ident: y
                                   Qualifier: AbsentTreeNode (leaf)
                                  END NameNode
                           END ObjectNode
                  END AssignNode
            END ExprStmtNode
           END list
          END BlockNode
    ReturnType: VoidTypeNode (leaf)
   END MethodDeclNode
   FieldDeclNode
    DefType: IntTypeNode (leaf)
    Modifiers: 1
    Name: NameNode
           Ident: x
           Qualifier: AbsentTreeNode (leaf)
          END NameNode
    InitExpr: AbsentTreeNode (leaf)
   END FieldDeclNode
  END list
  Modifiers: 0
  Name: NameNode
         Ident: OneFM
         Qualifier: AbsentTreeNode (leaf)
        END NameNode
  SuperClass: AbsentTreeNode (leaf)
 END ClassDeclNode
END list
Imports: <empty list>
```

```
 Pkg: NameNode
       Ident: test
       Qualifier: NameNode
                   Ident: java
                   Qualifier: NameNode
                               Ident: lang
                               Qualifier: NameNode
                                           Ident: ptolemy
                                           Qualifier: AbsentTreeNode
(leaf)
                                          END NameNode
                                END NameNode
                  END NameNode
      END NameNode
END CompileUnitNode
```

which is the representation for the following source code:

```
package ptolemy.lang.java.test;

class OneFM {
  public OneFM() {
    x = 1;
  }

  public int get() { return x; }

  public void set(int y) { x = y; }

  public int x;
}
```

To be more clear, the following is a graphical representation of the `set()` method in the above class:



5.2 Static Semantic Analysis

After parsing an input source file, the next step is to perform static semantic analysis on the AST returned by the parser. Mostly, static semantic analysis is the resolution of names to the declarations to which the names refer. We say resolution has been performed on a node in the AST if the contained NameNode has its DECL_KEY property pointing to the corresponding declaration.

The static class StaticResolution provides methods for performing static semantic analysis. Static semantic analysis is divided into three passes:

1. Pass 0: Package resolution, done by PackageResolutionVisitor consists of three steps:
    a. Creation of type environments, done by ResolvePackageVisitor. These environments are members of class declarations (ClassDecl), which are created during this step.
    b. Resolution of imports, done by ResolveImportsVisitor.
    c. Resolution of type names, done by ResolveTypesVisitor.

    Additional classes may be read in during pass 0.
2. Pass 1: Building of class and interface environments, done in two steps:
    a. Adding proper class and interface members to the respective environments, done by ResolveClassVisitor. All source files known to the compiler must undergo this step before the next step.
    b. Adding inherited class and interface members to the respective environments, done by ResolveInheritanceVisitor.

Additional classes may be read in during pass 1.

3. Pass 2: Resolution of names in statements, done in two steps:
    a. Resolution of references to local variables, parameters, and statement labels, done by ResolveNameVisitor.
    b. Resolution of fields, constructor calls, and method calls, done by ResolveFieldVisitor. Resolution of constructor calls and method calls requires that the types of the parameter expressions be known. These types are computed by TypeVisitor.

       Additional classes may **not** be read in during pass 2.

Static semantic analysis using the Titanium compiler and in this project requires that all the source code files for the classes encountered be available in the correct location relative to the CLASSPATH environmental variable. As a result, code that represents the class structure for the `java` package is required.

It is worth mentioning that in most cases full resolution only needs to be performed on a small subset of the source files that the compiler reads in. Pass 0 and 1 are required to resolve the class structure (contained constructors, methods, fields) of user types, and must be run on all referenced source files if a compile unit that uses (directly or indirectly) user types is to be pass 2 resolved. Pass 2, which is the resolution of names used in statements, does not need to be run on a CompileUnitNode if its statements are not of interest. For example, a user might want to perform pass 2 resolution on a source file that uses the class java.io.StringTokenizer, perhaps to do a later optimization pass. The source code for java.io.StringTokenizer would have to undergo pass 0 and pass 1 resolution, but since its statements are irrelevant to the user, it would not have to undergo pass 2 resolution.

As a result, pass 0 and 1 resolution are automatically performed on all files that have been read in if pass 2 is initiated on any file, but pass 2 is not automatically performed on any file.

In the case of a normal compiler, all of the files that are to be fully resolved are known from the start (they are usually specified on the command-line). However, in the case of this specialized code generation project, pass 2 may be invoked on the same file more than once. By caching the resolved AST after pass 2 is completed on a source file, the entire static semantic analysis procedure does not need to be redone. Actually, there are caches for passes 0, 1, and 2 that contain all ASTs that have undergone **at least** pass 0, 1, and 2 resolution, respectively. Thus invocation of `StaticResolution.load(String filename, int pass)` with 2 as the pass number argument, for instance, will check if the associated AST has undergone pass 2. If it has, `load()` returns the AST immediately. If not, it checks the cache for pass 1 resolved files. If a matching AST is found, `StaticResolution.resolvePass2()` is invoked on the AST, and the AST is added to the pass 2 resolved cache.

Though the caches are useful for improving performance, when code is transformed, they may need to be invalidated. This is done with the `StaticResolution.invalidateCompileUnit()` method.

A second optimization that was applied to the Java compiler is the use of **skeleton** code sources. A skeleton code source is a source file that has had the vast majority of statements removed, but retains the same public and protected classes, constructors,

methods, and fields so that its classes and interfaces may still be referenced by other source files. The use of skeleton code sources has the advantage that the time and memory required for lexical analysis, parsing, and static semantic analysis are greatly reduced over that of the original source file. Unfortunately, due to a subtle problem, some skeleton-ized source files are not legal Java, and will cause an error to occur if pass 2 is invoked on them. Nevertheless, as long as the statements in skeleton-ized source files are irrelevant to the application, pass 2 never needs to be invoked on skeleton source files. By default, the Java compiler favors skeleton files (with extension `.jskel`) over original source files. Skeletons can be generated by the standalone class Skeleton. Actually, instead of using the actual source code for the `java` package, skeletons are used to improve performance.

The standalone class Main may be used to do full static resolution of Java source files. After resolution is performed, the nodes in an AST are numbered with the visitor NumberNodeVisitor. The user may then select a node by its number, and inspect the node using ObjectInspector.

A feature of the Java compiler in Ptolemy II is that rules for determining types can be changed. This is accomplished by the use of the following classes:

- **TypeIdentifier** identifies types, and assigns an integer value to different kinds of TypeNodes.
- **TypePolicy** contains methods used to make decisions regarding types. It uses a contained instance of TypeIdentifier to identify types.
- **TypeVisitor** is an AST node visitor that uses its contained instance TypePolicy to determine the types of expressions. Types are lazily evaluated and cached in the value of the property TYPE_KEY, in nodes that represent expressions.

By default, the above three classes are used to do type resolution, but by providing an instance of a subclass of TypeVisitor to the method `setDefaultTypeVisitor()` in StaticResolution, the type "personality" of the compiler can be changed at runtime. One such personality is that of Extended Java, which knows about special Ptolemy math types and overloads operators between matrices, Ptolemy math types, and primitive types.

5.3 Java Compiler Back End

Unlike a traditional compiler, the Java compiler in Ptolemy II does not generate machine code, or even virtual machine code. Instead, Java source code can be regenerated from the AST by the visitor JavaCodeGenerator. This process is relatively trivial, and does not even use the information discovered by static semantic analysis. The reasoning behind this decision is that the purpose of the code generator in this project is not to translate code from one language to another, but to transform code in the same language, using information about the Ptolemy system.

The implementation of JavaCodeGenerator is a bit interesting. Each node visitation returns a **string list**, which is a list that may contain strings or other string lists. Each member of a string list represents a code fragment. Code fragments are assembled by building string lists instead of string concatenation. String concatenation only occurs when the entire string list is assembled for one source code file.

Used as a testing utility, the standalone class RegenerateCode takes a source file, parses it, and regenerates the source code from the AST.

Although regeneration of Java code is the only "back-end" to the Java compiler currently present in Ptolemy II, it should be a relatively straightforward task to write a visitor that converts the code into another language.

6. Generic Code Generation for Ptolemy II

The Ptolemy II code generator, found in the package `ptolemy.codegen`, uses the Java compiler to build a decorated AST, and then does transformations of the code for each actor. These transformations may be domain-independent, as in the case of token operations, or domain-specific, as in the case of port I/O. The overall code generation process consists of three passes:

1. Specialization of token declarations.
2. Transformations having to do with Ptolemy semantics into "Extended Java".
3. Conversion from Extended Java back into ordinary Java.

The class ActorCodeGenerator performs these passes with the help of some classes that may depend on the domain:

- **ActorCodeGeneratorInfo** contains information about the actor whose code is to be transformed. Depending on the domain, it may be extended to contain additional information.
- **PtolemyTypeIdentifier** identifies valid type names of actors, tokens, exceptions, parameters, and ports. It should be extended by domain-specific classes so that the different types of valid actors, exceptions, and ports in a domain are correctly identified. PtolemyTypeIdentifier extends the class TypeIdentifier.
- **PtolemyTypePolicy** is used to make decisions regarding types in Ptolemy, and it in turn uses an instance of PtolemyTypeIdentifier, which it contains. PtolemyTypePolicy extends the class TypePolicy.
- **PtolemyTypeVisitor** computes the types of expressions, and extends TypeVisitor, which is the ordinary visitor for computing types. PtolemyTypeVisitor ensures that the type of certain token operations is resolved to the most specific actual return type instead of the type declared by the method signature. For example, the method `add()` in the class DoubleToken is declared to have return type Token, but the method will return an instance of DoubleToken if it is called with an IntToken parameter. PtolemyTypeVisitor would then resolve the type of the method call expression to be DoubleToken instead of Token. This behavior is necessary to provide more detailed type information to SpecializeTokenVisitor. Each instance of PtolemyTypeVisitor contains an instance of PtolemyTypePolicy, and uses the instance of PtolemyTypeIdentifier contained by it.
- **ActorTransformerVisitor** does the transformation of step 2 above. The base class can do transformations of token operations, but not port I/O calls. Such calls must be transformed by subclasses of ActorTransformerVisitor, which depend on the domain.

Domain-specific behavior by the domain-independent class ActorCodeGenerator is achieved by the use of an Abstract Factory [5]. Each domain provides a subclass of CodeGeneratorClassFactory, which creates (possibly subclassed) instances of the above classes. Each subclass of CodeGeneratorClassFactory provides a consistent set of classes that may mutually assume each other's existence. For example, SDFTransformerVisitor, which extends ActorTransformerVisitor, contains a constructor that takes as an argument an instance of ActorCodeGeneratorInfo. It may assume that this argument is also an instance of SDFCodeGeneratorInfo so it can access the SDF-specific information in it. ActorCodeGenerator is given an instance of CodeGeneratorClassFactory, and uses the factory to create classes on an as-needed basis.

Before ActorCodeGenerator can begin transforming code, domain-specific information must be gathered about each actor. Therefore, the main code generation starting point is left for domain-specific classes. After domain-specific information is gathered, ActorCodeGenerator gathers domain-independent information and stores it in instances of ActorCodeGeneratorInfo, one instance per actor. The domain-specific main code generation class drives the three passes of transformations by invoking methods in ActorCodeGenerator.

ActorCodeGenerator generates code for each instance of an actor in a composite actor. Actor code, in general, cannot be shared even if two or more actors are instances of the same class. This is because the input and output ports are connected to different ports, and may be of different type. For each actor, the code for all classes that the actor class deeply extends, up to but not including the actor classes that are known in a specific domain, needs to be transformed. For example, consider the actor class RaisedCosine, which extends the class FIR, which extends the class SDFAtomicActor. In the SDF domain, SDFAtomicActor is a known actor class. As a result, the source code for both RaisedCosine and FIR would need to be transformed. By transforming a set of actor classes, the code for each actor class can still reference fields and methods of its superclasses. A more efficient, but more complicated alternative would be "flattening" all classes, i.e. putting all of the methods and fields of the set of classes into one class. Class flattening was not implemented in this project.

Classes are renamed so that there are no name conflicts between the code for each actor class and its subclasses. All instances of actors in a Ptolemy system have a unique name, which is used to name the classes of the transformed code. The format of the name is

"CG_" + <Class name> + "_" + <Ptolemy name>

In the previous example, if the Ptolemy name of an instance of RaisedCosine was "channel", two source files named `CG_RaisedCosine_channel.java` and `CG_FIR_channel.java` would be generated.

6.1 Assumptions Made by the Code Generator

Throughout the code generator, the following assumptions are made:

- There are no topology changes (changes in the relation of ports to each other, removal or insertion of actors, etc.) made during execution.

- Parameters do not change value during execution. In the generated code, the value assigned to Parameters is the value of data contained by the token retrieved by calling `getToken()` on the Parameter after the `preinitialize()` method of the containing composite actor completes. Typically, this poses no limitation except for the Expression actor, which parses a string in the Ptolemy II expression language and produces tokens with different values during run-time.
- There is no aliasing of ports or parameters. For example if an array was declared containing ports, the code generator cannot handle operations to the port such as `portArray[2].get(0)`. Any operations to ports and parameters must be performed on the public field variables that represent the ports and parameters directly.
- Arrays of tokens are not supported (neither `Token[]` nor ArrayToken).

6.2 Specialization of Token Declarations

One goal of code generation is to use the resolved type of tokens to replace token operations with operations on the data that are contained by the tokens. For example, the sequence of statements

```
int x = doSomething();
IntToken t1 = new IntToken(x);
DoubleToken t2 = new DoubleToken(4.5);
t2 = (DoubleToken) t2.add(t1);
```

should be transformed into

```
int x = doSomething();
int t1 = x;
double t2 = 4.5;
t2 = (double) (4.5 + x);
```

This is relatively easy to do, because IntToken maps directly to int, and DoubleToken maps directly to double. In the final statement, the original type of t2 resolved to DoubleToken and the type of t1 resolved to IntToken, so the meaning of the `add()` method call is well defined. However, consider the sequence

```
Token t1 = input.get(0);
Token t2 = new DoubleToken(4.5);
output.put(0, t1.add(t2));
```

where `input` is an input port and `output` is an output port. The transformer cannot directly transform the two declarations because Token may map to any of the supported data types in Ptolemy. Moreover, because the resolved types of `t1` and `t2` are both Token, the meaning of `add()` method call is unknown. It might be ordinary arithmetic addition, matrix addition, or even string concatenation.

However, since the types of the ports are known, a more specific type can be inferred for `t1` and `t2`. For example if input is of Ptolemy type `BaseType.INT`, the type of `t1` can be specified to IntToken. The type of `t2` can be specified to DoubleToken

by examining the assignment to a new instance of DoubleToken. So the code sequence can be transformed into

```
IntToken t1 = input.get(0);
DoubleToken t2 = new DoubleToken(4.5);
output.put(0, t1.add(t2));
```

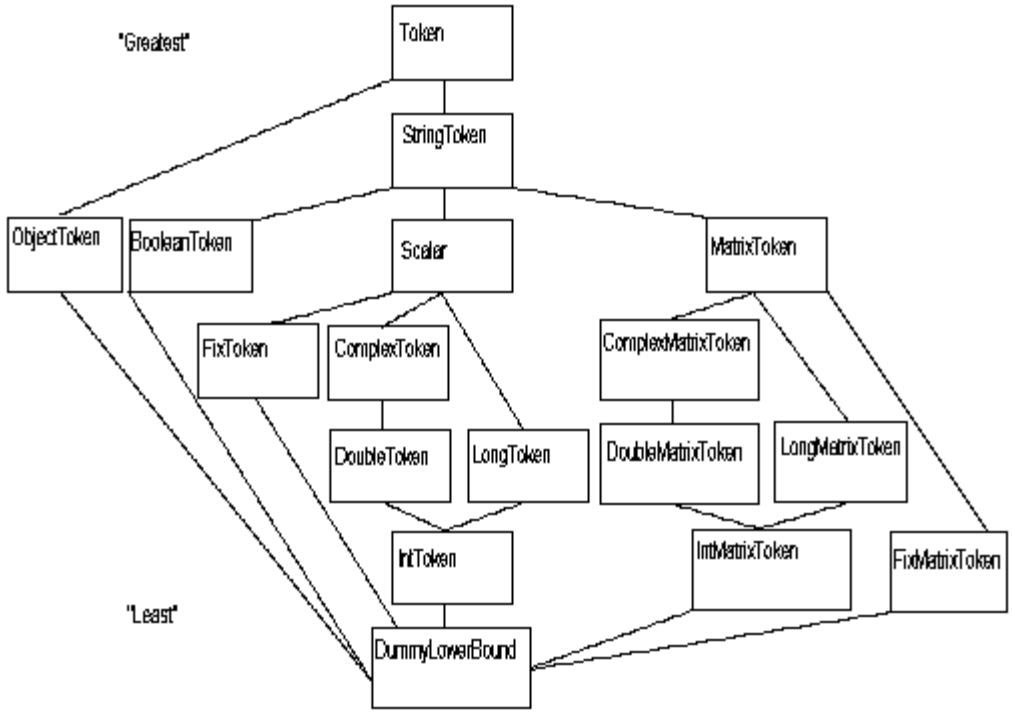
which later (in pass 2) can be further transformed as in the first example.

It is the task of the first pass is to specialize declarations of tokens. The above example is trivial; more sophisticated analysis needs to be done in the general case. For example, consider the following code sequence:

```
Token t1 = input1.get(0);
Token t2 = input2.get(0);
Token t3 = t1.add(t2);
```

Here the specialization of `t3` requires knowing the specialized types of `t1` and `t2`. In general, each expression of type Token or its subclasses can be represented as a node in a directed graph. Assignments, for example, would add edges from the right hand side to the left hand side, indicating that the left hand side is at least as general as the right hand side. The general solution can be obtained by representing the specialized type of each declared token as a variable. Constraints are put on these variables, and using a partial ordering in the form of a type lattice, the variables are solved together for the most specific types allowable. This is the same approach used to resolve the types of ports in Ptolemy, but applied instead to declarations of tokens. The implementation uses the same set of classes to solve inequalities as are used in Ptolemy. These classes are contained in the `ptolemy.graph` package.

The type lattice for token declarations differs slightly from the type lattice for port types:



In this modified type lattice, nodes represent types that can actually be used in declarations. Therefore, MatrixUpperBound, MatrixLowerBound, and Numerical from the original type lattice are omitted. The top of the lattice represents the most general type; the bottom of the lattice represents the least general type. In between the top and the bottom is a partial order in which any two nodes have a lower bound and an upper bound.

The implementation of token specialization uses a visitor for the AST, SpecializeTokenVisitor. First, variable terms are created for all declarations. Then, constraints are added to an InequalitySolver. The return value for each type of expression AST node is optionally an InequalityTerm that may be used as one of two InequalityTerms needed to form an Inequality. After all InequalityTerms have been added to the InequalitySolver for an actor class and the actor classes it inherits from, the InequalitySolver solves for the most specific type allowable for all variables.

The overall assumption for this scheme to work is that the most specific type of a Token variable remains constant. For example, a legal, concrete token type for `t1` cannot be solved for in the following legal code fragment:

```
Token t1 = new DoubleToken(4.5);
Token t2 = new IntMatrixToken(new int[1][1] {{42}});
t1 = t2;
```

because `t1` changes from a DoubleToken to an IntMatrixToken, which are **incomparable** in the type lattice (their common lower bound is DummyLowerBound). The most specific token type for `t1` would therefore be Token, which is not a concrete type.

As mentioned previously, SpecializeTokenVisitor uses an instance of PtolemyTypeVisitor to get more specific type information on return values of operations such as `add()`. In addition, SpecializeTokenVisitor also uses the information in ActorCodeGeneratorInfo to get the types involved in port I/O calls and the types of tokens contained in Parameters. To identify what classes are tokens, parameters, and ports, SpecializeTokenVisitor uses an instance of PtolemyTypeIdentifier.

The following rules are used to form inequalities:

Description	Example	Return value	Inequalities added
Declarations	α *t*;	N/A	type(*t*) ≤ α
Assignments	*t1* = *t2*	type(*t1*)	type(*t2*) ≤ type(*t1*)
Casts	(α) *t2*	type(*t2*)	type(*t1*) ≤ α
Method calls	*t1*.*op*(*t2*)	moreGeneral(type(*t1*), type(*t2*))	type(*t1*) ≤ retval type(*t2*) ≤ retval
	t1.zero(), *t1*.one()	type(*t1*)	
	t1.convert(*t2*)	type(*t1*)	type(*t2*) ≤ type(*t1*)
	param.getToken()	type(contained token)	
	port.get(*chan*)	type(*port*)	
	port.send(*chan*, *t*);	N/A	type(*t*) ≤ type(*port*)
	port.broadcast(*t*);	N/A	type(*t*) ≤ type(*port*)

α is any of the token types in the type lattice.
t, *t1*, *t2* are expressions with type Token or its subclasses that appear in the modified type lattice.
op ∈ {`add`, `subtract`, `multiply`, `divide`, `modulo`, `addReverse`, `subtractReverse`, `multiplyReverse`, `divideReverse`, `moduloReverse`}.
port is a public port of the actor.
chan is an expression with integer type, representing the channel number.
param is a public parameter of the actor.

After all these inequalities are added to the InequalitySolver, the "least solution" is found. The least solution is the solution in which variable terms are given a type as specific as possible, i.e. as close to the bottom of the type lattice as possible. It is possible that the least solution for some declarations is DummyLowerBound. This typically happens when a variable is declared but never actually assigned (assignments to `port.get(channel)`, where `port` is an unconnected port add no constraints – they will be eliminated in pass 2). In such a case, the declaration type is converted to Token, allowing error-free static resolution in preparation for the next pass. Assignments and operations involving the variable will be fixed in pass 3.

A limitation of the implementation of SpecializeTokenVisitor is that the relationship of a token being the element of a collection is not expressible as a normal

inequality, and so the information that can be inferred from such relationships is not used. For example, if `t2` is an instance of a subclass MatrixToken, the expression

```
t1 = t2.getElementAsToken(row, col)
```

can be used to infer the type of t1 if the specialized type of t2 is known to be, for instance, DoubleMatrixToken. Then type `t1` would be known to be DoubleToken, if the containment relationship could be expressed. It would be possible to infer the type of `t1` after the type of `t2` is found, but no constraint on the type of t2 would be considered, if for example, `t1` was used as a DoubleToken.

Another, probably more deleterious, example is the use of the `getArray()` method in SDF, which writes an array of tokens of the same type as the port:

```
Token[] tokenArray = new Token[len];
port.getArray(channel, tokenArray);
```

Even if the type of port is known, the above declaration cannot be specialized by the current implementation of SpecializeTokenVisitor. To solve these problems, the notion of containment could be added to InequalitySolver.

6.3 Transformations of Ptolemy Semantics

After token declarations are specialized, there should be no abstract Token declarations remaining in the AST for each actor. At this point, we can begin to do transformations of Ptolemy semantics. During transformation, only subclasses of supported actor classes are transformed.

So far, we have not described how transformation actually works. During normal replacement, when a node is visited, its return value is set to whatever node will take its place. That return value then replaces the node in its parent after visitation is complete. However, a more robust strategy is required for the removal of nodes and more complicated replacement of nodes. In the transformation process for this project, expressions embedded inside expression statements may signify that the expression statement should be replaced with a different statement. For example, in SDF, the expression statement

```
out.broadcast(t);
```

needs to converted to a `for` loop that iterates over the channels of `out`. `visitMethodCallNode()` is allowed to return an instance `ForNode`, although it is not even an expression. Based on the return value of `accept()` on the embedded expression node, ExprStmtNode may either replace the embedded expression with its return value and return itself, or return the return value of the embedded expression node. A more general replacement strategy that allows such functionality, implemented in this project, can be summarized as follows:

- A visitation of a member node of a class or interface may return a member node, `NullValue.instance`, or a list of member nodes. A return value of `NullValue.instance` indicates that the member is to be removed.
- A visitation of a MethodCallNode to a method with return type `void` may return an expression node, `NullValue.instance`, a statement node, or a list of objects. A return value of `NullValue.instance` indicates that the expression statement containing the expression is to be removed. A return value of a statement node signifies that expression statement node containing the expression node should be replaced with the returned statement node. A return value of a list of objects may contain expression nodes, statements, or `NullValue.instance`. Each expression node in the list is checked if it represents an expression that may have side effects. If so, it is placed in an ExprStmtNode. The list of objects is thus converted to a series of statements. These statements are wrapped in a BlockNode, which replaces the expression statement node that contains the MethodCallNode.
- A visitation of a ExprStmtNode node must handle the return value of the embedded expression node (which may be a MethodCallNode) appropriately.
- A visitation of a statement node may return a statement node, `NullValue.instance`, or a list of objects. A return value of `NullValue.instance` indicates that the statement is to be removed. A return value of list of objects is converted to a BlockNode as above, which replaces the statement node.
- Visitation of nodes that contain statement nodes must handle the return value of the embedded statement nodes appropriately.

To preserve side effects for all method calls, a more robust replacement strategy would allow all expressions to return lists of objects. For example, consider the statement

```
ht = out.hasToken(chan = x);
```

which, in SDF, is transformed into

```
ht = true;
```

if `out` is a connected port. If the expression `chan = x` is not preserved, its side effect will be lost. In general, a MethodCallNode (which is a type of an expression node) may contain a list of more than one argument expressions, and these expressions may each have side effects. The AssignNode that contains the MethodCallNode must propagate these side effects up to the ExprStmtNode that contains it. Therefore, to preserve these side effects, all expressions nodes must be able to return lists of expressions nodes that can be converted to statements nodes that follow the expression statement node that contains the expression node. In this project, we have not yet attempted to do this; code that contains such side effects is rare.

Now that we have a replacement strategy that more or less works, let us consider transformations of token declarations and operations. Each token declaration is

transformed into a declaration of the type of the data contained by the token. For instance,

```
IntToken t;
```

is transformed into

```
int t;
```

but Token variables that could not be specialized in pass 1 remain type Token in pass 2.

Creation of new tokens is replaced by the value that is contained by the created token. For instance,

```
double[][] z = {{1.0, -1.0}, {4.3, 0.5}};
x = new DoubleToken(z);
```

is transformed into

```
double[][] z = {{1.0, -1.0}, {4.3, 0.5}};
x = z;
```

Access of the data contained by tokens is transformed into a reference to the former token variable. For instance,

```
IntMatrixToken im = new IntMatrixToken(new int[][] {{5, 7}});
double[][] dm = im.doubleMatrix();
```

is transformed into

```
int[][] im = new int[][] {{5, 7}};
double[][] dm = im;
```

This is not legal Java; a conversion from `int[][]` to `double[][]` must be added in the next pass.

Most operations on tokens are transformed into ordinary arithmetic expressions. For instance,

```
a = b.subtract(c);
```

is transformed into

```
a = b – c;
```

Note that b and c may be expressions of non-primitive type. If, for example, `b` and `c` are both of the type Complex, the above statement would not be legal in ordinary Java. Such statements are transformed again in the next pass.

The methods `one()`, `oneRight()`, and `zero()` methods must be dealt with in a special way. Given a DoubleToken type `t1`,

```
DoubleToken t2 = t1.one();
```

is transformed into

```
double t2 = 1.0;
```

but if `t1` is a matrix or fixed point token type, a special static method in the `ptolemy.math` package must be called. For instance,

```
DoubleMatrixToken t2 = t1.one();
```

is transformed into

```
double[][] t2 = CodeGenUtility.one(t1);
```

where `CodeGenUtility.one()` is a static method that returns an identity matrix with the same number of rows as the argument matrix. It is overloaded for all matrix types and fixed point numbers. `oneRight()` and `zero()` are similarly transformed.

Now let us consider transformations dealing with parameters. Parameters are public fields of actors. These fields are replaced with the value contained by the token returned by calling `getToken()` on the parameter after the preinitialization phase is completed. For instance, consider the following field declaration:

```
public Parameter pmf;
```

If the parameter value is set to a DoubleMatrixToken containing a matrix of doubles {{0.65, 0.35}}, the code is transformed into

```
public final double[][] pmf = {{0.65, 0.35}};
```

When the `getToken()` method call is encountered, it is transformed into a reference to the parameter. For instance,

```
DoubleToken pmfToken = pmf.getToken();
```

is transformed into

```
double[][] pmfToken = pmf;
```

Attempts to set the types or values of parameters are discarded, because it is assumed that parameters do not change value during run-time.

During the course of the simulation of a Ptolemy model, certain exceptions may be thrown. These exceptions are typically not thrown unless an error is made in the usage of an actor. Therefore, in pass 2 of the transformer, all statements that throw Ptolemy exceptions are converted into statements that throw run-time exceptions. All clauses that

catch Ptolemy exceptions are discarded. Finally, Ptolemy exceptions that appear in the declared list of exceptions that a method or constructor may throw are removed.

Method calls to ports and actors are also performed in this pass, but the transformations are domain-specific. ActorTransformerVisitor provides the following methods that subclasses may override to perform domain-specific transformations:

- `_actorClassDeclNode()` is called when the class declaration node of the actor class is found.
- `_actorMethodCallNode()` is called when a node representing a method call to an actor is found.
- `_portFieldDeclNode()` is called when a node representing a field declaration of a port variable is found.
- `_portMethodCallNode()` is called when node representing a method call to a port is found.

These methods are provided to encourage reuse of the code in ActorTransformerVisitor, but actually, any visitation method in ActorTransformerVisitor can be overridden.

Finally, there are a few miscellaneous transformations that ActorTransformerVisitor performs. One transformation worth mentioning is that the actor class hierarchy must be cut at the point where an actor class inherits from a known actor class in a domain. Again consider the example of the actor class RaisedCosine, which extends the class FIR, which extends the class SDFAtomicActor. The transformed class CG_FIR_channel would extend java.lang.Object instead of SDFAtomicActor. Calls to `super.fire()`, etc. in CG_FIR_channel need to be removed. Also, default execution methods `preinitialize()`, `initialize()`, etc. need to be added to CG_FIR_channel if they do not already exist.

6.4 Conversion of Extended Java to Java

"Extended Java" is the name we have given to a superset of Java that has extended type rules dealing with complex numbers, fix point numbers, and two-dimensional matrices. Just as a integer may be "widened" to a double in Java, a matrix of integers can be "widened" to a matrix of doubles in Extended Java. Just as an integer can be added to a double in Java, a matrix of integers can be added to a matrix of complex numbers in Extended Java. The type rules follow the Ptolemy type lattice.

Extended Java would be a very convenient programming language, especially for scientific purposes, but because there are no Extended Java compilers, we revert Extended Java ASTs back to ordinary Java ASTs in pass 3 of code generation. The conversion process is a straightforward case analysis of expressions and their types. In order to convert matrices into different types of matrices and to perform operations among complex numbers, fix point numbers, and matrices, nodes that represent method calls to methods in classes in the `ptolemy.math` package are created quite frequently.

Another task of pass 3 is to transform nonsensical code resulting from token variables that cannot be specialized by SpecializeTokenVisitor. Those variables are declared as type Token, which is changed in pass 3 to Object. Any assignment to such variables is converted to an assignment from `null`.

Expressions that involve operands of type Token need to be converted to legal, ordinary Java. Because these expressions are never actually executed (they are embedded

inside pieces of dead code), it is acceptable to convert all such operands to any expression that satisfies the type rules of ordinary Java. In this project, a consistent dummy value is chosen based on the target type.

To illustrate what pass 3 does, consider the following `fire()` method of the AddSubtract actor. This particular instance of the actor is responsible for adding two double matrices, which are taken from the `plus` port. The `minus` port is unconnected so this instance does not actually perform subtraction. After pass 2, the `fire()` method is transformed into

```
public void fire() {
    double[][] sum = null;
    for (int i = 0; i < 2; i++) {
        if (true) {
           if (sum == null) {
              sum = this._cg_plus_chan_buffer[this._cg_chan_temp_r = i]
               [this._cg_plus_offset[this._cg_chan_temp_r] =
                (this._cg_plus_offset[this._cg_chan_temp_r] + 1) %
                this._cg_plus_chan_buffer_len[this._cg_chan_temp_r]];
           } else {
             sum = sum +
             this._cg_plus_chan_buffer[this._cg_chan_temp_r = i]
              [this._cg_plus_offset[this._cg_chan_temp_r] =
               (this._cg_plus_offset[this._cg_chan_temp_r] + 1) %
               this._cg_plus_chan_buffer_len[this._cg_chan_temp_r]];
              // not legal Java
           }
        }
    }

    // the following loop is the result of the unconnected minus port
    // it's all dead code
    for (int i = 0; i < 0; i++) {
        if (false) {
           Token in = null;
           if (sum == null) {
              sum = null;
           }

           sum = sum - in; // not legal Java
        }
    }
}
```

(The complicated expressions involving `cg_plus_chan_buffer` are the result of the SDF transformation of `plus.get(i)`. See the next section for details.) After pass 3, the `fire()` method is transformed into

```
public void fire() {
    double[][] sum = null;
    for (int i = 0; i < 2; i++) {
        if (true) {
           if (sum == null) {
              sum = this._cg_plus_chan_buffer[this._cg_chan_temp_r = i]
```

```
                [this._cg_plus_offset[this._cg_chan_temp_r] =
                 (this._cg_plus_offset[this._cg_chan_temp_r] + 1) %
                 this._cg_plus_chan_buffer_len[this._cg_chan_temp_r]];
            } else {
              sum = DoubleMatrixMath.add(sum,
              this._cg_plus_chan_buffer[this._cg_chan_temp_r = i]
               [this._cg_plus_offset[this._cg_chan_temp_r] =
                (this._cg_plus_offset[this._cg_chan_temp_r] + 1) %
                this._cg_plus_chan_buffer_len[this._cg_chan_temp_r]]);
            }
        }
    }

    for (int i = 0; i < 0; i++) {
        if (false) {
           Object in = null;
           if (sum == null) {
              sum = null;
           }

           sum = DoubleMatrixMath.subtract(sum, null);
        }
    }
}
```

Note that the `+` operator is overloaded between double matrices in Extended Java, and the operation was converted to a call to `DoubleMatrixMath.add()`. Also note that `in`, for which a specific token type could not be found, was transformed to `null` so that it can be passed as an argument to `DoubleMatrixMath.subtract()`.

An additional step to handle expressions involving unspecified token variables might be to convert any expression involving a Token operand into a dummy value. In the above example, `DoubleMatrixMath.subtract(sum, null)` might as well be replaced with `null`. Morever, the assignment to `sum` might as well be eliminated. However, since the code is never executed, and could be eliminated by an optimizing compiler, such a step is not really necessary.

The code that does conversion from Extended Java to Java is found in the `ptolemy.lang.java.extended` sub-package. To compile a program written in Extended Java, the type personality of the compiler must be changed to use instances of ExtendedJavaTypePolicy and ExtendedJavaTypeIdentifier. ExtendedJavaConverter can then transform a resolved AST that follows the Extended Java type rules to ordinary Java.

7.0 Code Generation for SDF

In the synchronous dataflow (SDF) domain, actors can be run sequentially and read and write tokens from/to fixed sized, circular, FIFO buffers. The order in which actors execute is determined by the scheduler. A schedule can be computed because each actor declares rates at which it receives and sends tokens. In general, there is more than one valid schedule for a SDF system. Different schedules may result in better code [13], but in this project we have focused on generating code given a fixed schedule. However, we note that the current schedule is flattened, i.e. it supports no loops. For example, the

looped schedule A(2 BC)(2 C), where A,B, and C are actors is flattened to ABCBCCC in Ptolemy II. It is likely that using looped schedule would result in more efficient generated code compared to the flattened schedule.

This implementation of the SDF code generator assumes that each input channel is connected to at most one output channel.

The first step of the SDF code generator is to figure out the size of the required buffers. In this implementation, a one-dimensional array is allocated for each output port that has only one channel. A two-dimensional array is allocated for each output port with more than one channel, with the number of rows equal to the number of channels. The length of each one-dimensional array (or row of each two-dimensional array) is conservatively set to

(initial production) + (appearances in schedule) * (tokens produced per appearance)

Here "initial production" refers to the number of tokens produced before execution according to the schedule begins. The number of "appearances" refers to the number of times the actor containing the port appears in the schedule. The buffer is used circularly, so increasing offsets into the buffer must be taken modulo the buffer size.

Buffers are public, static fields of the main class, CG_Main, so that they may be accessed for input and output by the transformed actors. If an output port `out` has type LongMatrixToken, 5 channels, and the buffer length is computed to be 10, the following field declaration will appear in CG_Main:

```
public static final long[][][][] _cg_out_2 = new long[5][10][][];
```

The "2" in `_cg_out_2` is an artifact of the buffer namer, which ensures names do not conflict.

Since we assume that each input channel is connected to at most one output channel, there is one one-dimension array used to store the data for each output channel to input channel connection. For each channel, the position in the buffer must be stored. Actually, the last position read from / written to is used to simplify transformations (the update of the position can be done without additional statements). If a port `out` with one channel is present, the following field is added to the actor class:

```
protected int _cg_out_offset = -1;
```

If a port `out` with three channels is present, the following field is added to the actor class:

```
protected int[] _cg_out_offset = {-1, -1, -1};
```

With the previous variables, the `send()` method called on a port with one channel can be transformed from

```
out.send(0, t);
```

to

```
CG_Main._cg_out_2[_cg_out_offset = (_cg_out_offset + 1) % 10] = t;
```

and in the case of a port with more than one channel,

```
out.send(1, t);
```

is transformed into

```
CG_Main._cg_out_3[1][_cg_out_offset[1] =
 (_cg_out_offset[1] + 1) % 10] = t;
```

In the case in which the channel number is not constant (constant folding helps, especially after loops are unrolled), a temporary variable must be used to avoid side effects. So

```
out.send(whatCh(), t);
```

is transformed into

```
CG_Main._cg_out_3[_cg_chan_temp_w = whatCh()]
 [_cg_out_offset[_cg_chan_temp_w] =
  (_cg_out_offset[_cg_chan_temp_w] + 1) % 10] = t;
```

Transformation of the method call `broadcast()`, which writes the same token to all channels of an output port, is transformed by placing similar statements in a loop. Thus,

```
out.broadcast(t);
```

is transformed into

```
for (int _cg_chan_temp = 0; _cg_chan_temp < 3; _cg_chan_temp++) {
  CG_Main._cg_out_3[_cg_chan_temp][_cg_out_offset[_cg_chan_temp] =
   (_cg_out_offset[_cg_chan_temp] + 1) % 10] = t;
}
```

if `out` is an output port with three channels.

For input ports, the buffer associated with each channel is not contained in the same two-dimensional array. To support channel expressions that are not constant, a table of buffers associated with each channel is required to resolve which buffer from which to read. In addition, a table of the lengths of the buffers is required. These tables are added to actor classes, and may look like:

```
protected final double[][][][] _cg_in_chan_buffer =
 {CG_Main._cg_output_4, CG_Main._cg_output_5};

protected final int[] _cg_in_chan_buffer_len = {16, 32};
```

where `in` is an input buffer with two channels. Now,

```
in.get(0)
```

can be transformed into

```
CG_Main._cg_output_4[_cg_plus_offset = (_cg_plus_offset + 1) % 16]
```

if `in` is a port with one channel, whose associated buffer length is 16.

In the case of a port with multiple channels, if the channel expression is constant, and the associated buffer and buffer size are known at transformation time. Then,

```
in.get(1)
```

can be transformed into

```
CG_Main._cg_output_5[_cg_in_offset[1] =
     (_cg_in_offset[1] + 1) % 32]
```

If the channel expression is not a constant, we have our most complicated transformation, which needs to lookup the buffer and buffer size associated with the channel while avoiding side effects:

```
in.get(whatCh())
```

is transformed into

```
_cg_in_chan_buffer[_cg_chan_temp_r = whatCh()]
     [_cg_in_offset[_cg_chan_temp_r] =
      (_cg_in_offset[_cg_chan_temp_r] + 1) %
       _cg_in_chan_buffer_len[_cg_chan_temp_r]]
```

In the case where the channel expression is not constant, the transformed code seems quite complicated. However, such code typically occurs in loops where the channel number increases uniformly between two known values. If such loops are unrolled by an optimizing compiler, the code can be reduced in complexity to that of constant channel numbers.

A number of optimizations might be applied to the method in which buffers are allocated and used. If the buffer sizes are constant among a connected group of input and output channels, there is no need to look up the length of the buffer. If buffer sizes are rounded to the next power of two, costly modulo operations can be replaced with cheaper bitwise AND operations. If the buffer size is one, the same location in the buffer is always used, and no update of the offset into the buffer is required. As of yet, none of these optimizations have been implemented in this project.

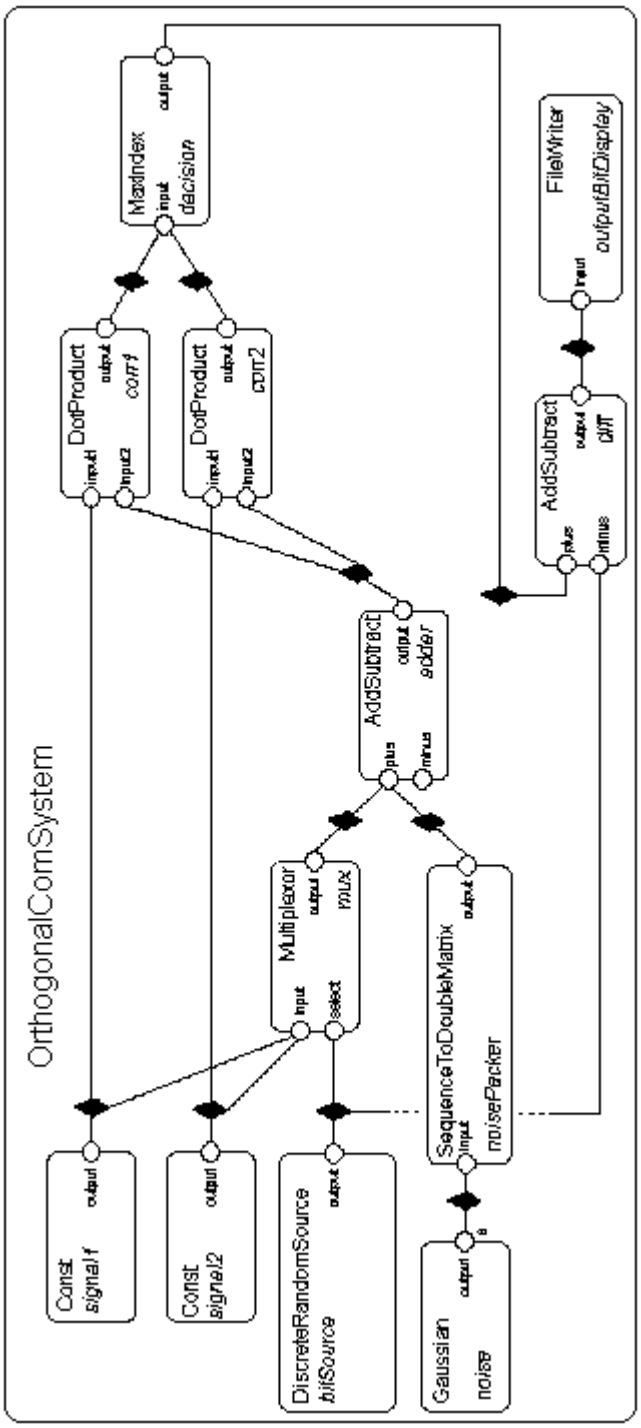
The generation of the `main()` method in SDF simply places sequential method calls to `preinitialize()`, `initialize()`, etc. according to Ptolemy semantics and the scheduler.

The above code generation process is performed with the following classes, found in the `ptolemy.domains.sdf.codegen` sub-package:

- **SDFCodeGenerator** is the top-level class which gathers SDF specific information for each actor and places it in instances of SDFActorCodeGeneratorInfo. It then drives the three passes of ActorCodeGenerator, creating the CG_Main class between the second and third pass.
- **SDFActorCodeGeneratorInfo** extends ActorCodeGeneratorInfo, and contains information about buffers and the number of times an actor appears in the schedule.
- **SDFTypeIdentifier** identifies the additional port type and actor type found in SDF, and extends PtolemyTypeIdentifier.
- **SDFActorTransformerVisitor** transforms port method calls as described above, and extends ActorTransformerVisitor.
- **SDFCodeGeneratorClassFactory** extends CodeGeneratorClassFactory and creates instances of SDFActorCodeGeneratorInfo, SDFTypeIdentifier, and SDFActorTransformerVisitor to be used by ActorCodeGenerator.

8. Example system

SDF code generation was successfully performed on the following system, which does the transmission and detection of discrete-time, orthogonal signals. One of two orthogonal signals is selected based on the input bit, which is generated by a Bernoulli process with p = ½. The signal is then corrupted by additive white Gaussian noise before reception. The received signal is correlated with the two orthogonal signals by performing the dot product operation. The waveform with the maximum correlation corresponds to the decision made on which waveform was transmitted. The following is the block diagram of the system:



signal1 and *signal2* output double matrices representing eight samples of a signal. All ports in the diagram have a consumption or production rate of one, except for the output port of the Gaussian noise source. The noise samples are packed into a matrix by the *noisePacker* actor. The difference between the input bit and the output bit is printed to standard output by the *outputBitDisplay* actor.

This composite actor is a reasonably good test case for the code generator because it tests the following features:

- Input from multiple channels
- Output to multiple destinations
- Non-unity token production rate of output port
- Addition of DoubleMatrixTokens
- Unconnected port
- Use of both domain-polymorphic and SDF-specific actors

All calls to the Ptolemy `actor`, `kernel`, `data`, and `domains.sdf.kernel` packages were eliminated. However, additional dependencies on the Ptolemy math package were added due to the addition of DoubleMatrixTokens.

The code generated version exhibits a significant performance improvement over the original simulated version, especially in the latency to get the first output sample. Normal iterations also execute faster. In general, the performance improvement of a code generated system versus the simulated system will depend on the ratio of time spent performing token and I/O operations to the total execution time.

9. Lessons Learned

From a compiler builder's perspective, we learned that the Visitor pattern provides a simple, modular alternative to the traditional architecture in which nodes contain code for doing all sort of operations. In addition, we learned that generating the node classes from a single definition file saves considerable effort, especially when the design of the node hierarchy is not stable. Now that the GenerateVisitor program has been completed, the savings of effort in the next compiler project should be even greater.

We also learned that the use of skeletons saves considerable memory and time during the compilation.

As Java programmers, we learned from this project to be careful using the implementations of collections in `java.util`. In our design, we use HashMaps and HashSets liberally; each node in the AST contains a HashMap and a HashSet, due to the fact that TreeNode extends PropertyMap and TrackedPropertyMap. By using the default constructors for HashMap and HashSet without specifying a capacity, extraneous space was allocated for buckets in the form of Object arrays. By default, the number of buckets seems to be 16, which is much more than the number of properties or visitors that a node could reasonably have. Before this problem was discovered (using the JProbe profiler tool), the memory used by Object arrays was 80% of the total memory used by the code generator. As a result, the code generator frequently ran out of memory and could not complete. The solution to this problem is trivial: specifying a smaller capacity to the constructors of HashMap and HashSet. With this fix, the memory used by Object arrays was reduced to 30% of the total memory. The total memory usage was reduced by a factor of two, allowing the code generator to complete without running out of memory.

10. Future Applications

There are a number of applications that may use directly use parts of the source code of this project. The `ptolemy.lang` package provides a number of classes that enable a programmer to quickly write a compiler for a different language. In addition, the GenerateVisitor program may be used to generate a visitor and tree nodes for that language.

A full Java compiler may be completed using the Java compiler front-end found in the `ptolemy.lang.java` package. Back-ends and optimizers could be easily implemented as visitors of a Java AST. Also, a number of utility programs might be implemented using visitors. An example of such a program is one that removes extraneous `import` statements from source code. For such an application, dealing with the resolved AST that represents the source code is probably the only good solution.

It would also be possible to implement a Java to C converter using visitors, but there are already a number of programs that do this conversion. A more interesting application would be to generate hardware descriptions (such as VHDL) that would perform the same actions as found in an AST.

Using the existing Java compiler, extensions to Java that simply change typing rules, such as Extended Java, could be further developed to ease scientific or engineering programming by overloading operations between special types like matrices. The purpose of Extended Java is just to be an intermediate language for code generation, but it can actually be used as a general-purpose programming language because it can be converted to ordinary Java.

The actor code transformer is probably limited to Ptolemy II systems because it assumes Ptolemy semantics throughout. However, within Ptolemy II, there are many more domains besides SDF that might benefit from code generation, so that code generated systems may be run without the Ptolemy II software infrastructure in memory or performance constrained environments.

The code generator for SDF might be modified to generate code executable in parallel processing environments. However, the existing scheduler for SDF would have to modified to find a suitable parallel schedule.

11. Obtaining the Source Code

It is the hope of the author that the code to do Java compilation and actor transformation for SDF will be included in the next release of Ptolemy II. At present, the code is part of the development tree. Current releases of Ptolemy II can be found at http://ptolemy.eecs.berkeley.edu/.

12. Acknowledgements

The work presented in the paper was funded by the Ptolemy project, headed by Professor Edward Lee. I am grateful to Professor Lee for his influence on the design of this project, as well as for supporting me as a researcher for two full years. I would also like to thank the other members of the Ptolemy group for their support and friendship.

13. References

[1] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong. "Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java". Memorandum UCB/ERL M99/44, EECS, University of California, Berkeley, July 19, 1999.

[2] J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development", vol.4, pp. 155-182, April, 1994.

[3] MathWorks. Real Time Workshop 3.0 Data Sheet. http://www.mathworks.com/products/rtw/9400v01.rtw3.pdf.

[4] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. "Titanium: A High Performance Java Dialect". ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California, February 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, ISBN 0-201-63361-2, October 1994.

[6] Vern Paxson. "Flex, Version 2.5: A Fast Scanner Generator". http://www.gnu.org/manual/flex-2.5.4/html_mono/flex.html. March 1995.

[7] Charles Donnelly and Richard Stallman. "Bison: The YACC-compatible Parser Generator". http://www.gnu.org/manual/bison-1.25/html_mono/bison.html. November 1995, Bison Version 1.25.

[8] Elliot Berk. "JLex: A Lexical Analyzer Generator for Java™". http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html. Department of Computer Science, Princeton University. October 29, 1997.

[9] Bob Jamison. "BYACC/J, Java Extension v 0.93". http://www.lincom-asg.com/~rjamison/byacc/. June 1, 1999.

[10] Sun Microsystems. "JavaCC v 1.1". http://www.metamata.com/JavaCC/. 1999.

[11] Terence Parr. "ANTLR Reference Manual". http://www.antlr.org/doc/index.html. January 19, 2000.

[12] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. Addison-Wesley, ISBN 0-201-63451-1, 1996

[13] Shuvra S. Bhattacharyya, Praveen K. Murthy, Edward A. Lee. “Synthesis of Embedded Software from Synchronous Dataflow Specifications”. Journal of VLSI Signal Processing 21, 151-166, 1999.

Appendix A: The GenerateVisitor Program

The GenerateVisitor program takes as input a node definition file describing all nodes of a language, and outputs the following in source code form:

- The base visitor class for the language.
- The classes for all nodes in the language.
- An interface containing node class identifiers for all nodes in the language.

GenerateVisitor uses the StringTokenizer class to scan the input file, so a complicated syntax is not possible.The following is the syntax of the node definition file:

```
// comments start with double slash

<cheader>
Common header for visitor, nodes, and interface containing class ID's.
</cheader>

<vheader>
Header for visitor,appended after the  common header.
</vheader>

<nheader>
Header for node types, appended after the common header.
</nheader>

<iheader>
Header for interface containing class ID's, appended after the common header.
</iheader>

// more comments here are ok
// between each node class description, white space and comments are ok

NodeClassDescriptions

<end of file>
```

Each node classes or interfaces is described with a *NodeClassDescription*, which has the following syntax:

NodeClassDescription -> Name FullNodeKind Name Implements$_{Opt}$ *Members*$_{Opt}$ <newline>

The desciption should contain no newline characters except for the final character.

The first *Name* is the name of the node class or interface.
The second *Name* is the name of the class or interface the node class or interface extends.

FullNodeKind describes the kind of node and whether or not it appears in the parse tree. It has the following syntax:

FullNodeKind -> *NodeKind*`N`$_{opt}$

If `N` appears at the end of *FullNodeKind*, the node type is assumed not to appear in the parse tree.

NodeKind describes the kind of node, and has the following syntax:

NodeKind -> `A` (abstract class)
-> `S` (singleton – automatically concrete)
-> `C` (concrete class)
-> `I` (interface)

A concrete class is assigned a integer class ID, which is added to the interface containing class IDs. Each concrete class is given a method

```
public final int classID();
```

The purpose of the class ID is to be used in `switch` statements that have `case`s that depend of the kind of node. Concrete classes that are assumed to be in the parse tree have a corresponding

`Object visit`*NodeName*`(`*NodeName*` node, LinkedList args);`

method in the visitor class for the language, and override the `_acceptHere(IVisitor v, LinkedList args)` method of TreeNode to call the visitation method.

Singletons are classes that have only one global instance, accessed by

NodeName`.instance`

Singletons have only a private constructor and may not contain members. They override the `isSingleton()` method of TreeNode, returning `true`.

The class or interface may implement zero or more interfaces:

Implements -> `i` *Names* `i`

Names are names of interfaces that the class directly implements, separated by spaces.

A class or interface may contain zero or more members:

Members -> *DefiningConstructor*

-> NormalConstructor
-> MethodStub

A defining constructor adds the constructor parameters to the child list of TreeNode if the argument is not a primitive Java type or String; otherwise the parameter is made an ordinary member of the node class. The following is the syntax of a defining constructor:

DefiningConstructor -> `k` *superArgNum Parameters*$_{Opt}$ `k`
Parameter -> Type Name

superArgNum is an non-negative integer.
Type is a string describing the type of the parameter.
Name is the name of the parameter.

The constructor passes *superArgNum* number of arguments to the `super()` constructor call; the rest of the arguments are added to the node class.

Each parameter up to and including the *superArgNum*th parameter must match those of the superclass exactly. For each parameter after the first *superArgNum*-1 parameters, the following access methods are added, with appropriate implementations:

```
Type getCapName();
void setCapName(Parameter);
```

where *CapName* is *Name* with the first letter capitalized.

A normal constructor is like a defining constructor, but does not add variables corresponding to parameter names to the class or access methods:

NormalConstructor -> `c` *superArgNum Parameters*$_{Opt}$ `c`

For a class, a method stub represents a method that returns a dummy value for the given return type. For an interface, no implementation of the method is required. The following is the syntax for a method stub:

MethodStub -> `m` *ReturnType Name Parameters*$_{Opt}$ `m`

ReturnType is the return type of the method, which may be `void`.
Name is the name of the method.
Parameters$_{Opt}$ are the optional parameters of the method, separated by spaces.

The following definition file illustrates the syntax:

```
<cheader>
/* Copyright Taco Software, 2000. */
</cheader>
```

```
<vheader>
package com.taco.toy;

// the following imports should always be included
import java.util.LinkedList;
import java.util.List;
import ptolemy.lang.IVisitor;
import ptolemy.lang.TreeNode;

// need to reference the output package for the nodes
import com.taco.toy.nodetypes.*;

/** A visitor for AST's for Toy code. */
</vheader>

<nheader>
// this should reference the package you put the nodes in
package com.taco.toy.nodetypes;

// the following imports should always be included
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import ptolemy.lang.IVisitor;
import ptolemy.lang.ITreeNode;
import ptolemy.lang.TreeNode;
import ptolemy.lang.java.JavaVisitor;

/** A specific type of node in the AST for Toy code. */
</nheader>

<iheader>
// this should reference the package you put the nodes in
package com.taco.toy.nodetypes;

/** An interface containing the class ID's of all concrete node types.
*/
</iheader>

// interfaces
NamedNode I ITreeNode m NameNode getName m m void setName NameNode name
m

NameNode C TreeNode k 0 String ident k

// abstract
TypeNode A TreeNode c 0 c
ExprNode A TreeNode c 0 c
BinaryArithNode A ExprNode k ExprNode expr1 ExprNode expr2 k
VarDeclNode A TreeNode i NamedNode i k 0 TypeNode defType NameNode \
 name k
LiteralNode A ExprNode k 0 String literal k

// literal
IntLitNode C LiteralNode c 1 String literal c

// type
```

```
IntTypeNode S TypeNode

// expressions
PlusNode C BinaryArithNode c 2 ExprNode expr1 ExprNode expr2 c
VarNode C ExprNode k 0 NameNode name k

// local variable declaration
LocalVarDeclNode C VarDeclNode k 2 TypeNode defType NameNode name \
 ExprNode initExpr k
```

In the above definition file, the \ character is used to indicate a continuation of the line, but this is not actually legal. Each node description must be contained in exactly one line.

Appendix B: Java AST Node Type Descriptions

Most of these descriptions were taken from Professor Paul Hilfinger's CS164 Java compiler class project, which seems to be the basis of the Titanium compiler.

Interfaces

NamedNode
  Extends: ITreeNode
  Attributes: NameNode name
 Represents:
  A TreeNode with a name.

ModifiedNode
  Extends: ITreeNode
  Attributes: int modifiers
 Represents:
  A TreeNode with a name.

StatementNode
  Extends: ITreeNode
 Represents:
  A statement.

Abstract Classes

TypeNode()
  Parent type: TreeNode
 Represents:
  The base type for all ASTs that represent types

PrimitiveTypeNode()
  Parent type: TypeNode
 Represents:
  The base type of all primitive types

ReferenceTypeNode()
  Parent type: TypeNode
 Represents:
  The base type of all reference types

VarDeclNode(int modifiers, TypeNode defType, NameNode name)
  Parent type: TreeNode
  Implements: NamedNode, ModifiedNode
 Represents:

Base type of declarations of a variable.

VarInitDeclNode(int modifiers, TypeNode defType, NameNode name,
TreeNode initExpr)
Parent type: VarDeclNode
Represents:
The base type of declarations of a variable with an initializer.

UserTypeDeclNode(int modifiers, NameNode name, List interfaces, List members)
Parent type: TreeNode
Implements: NamedNode, ModifiedNode
Represents:
The base type of declarations of user types (classes and interfaces).

InvokableDeclNode(int modifiers, NameNode name, List params, List throwsList)
Parent type: TreeNode
Represents:
The common base type for declarations of invokable entities (constructors and methods).

ConstructorCallNode(List args)
Parent type: TreeNode
Implements: StatementNode
Represents:
A constructor call with the argument list ARGS.

IterationNode()
Parent type: TreeNode
Implements: StatementNode
Represents:
Base class for loops.

JumpStmtNode(TreeNode label)
Parent type: TreeNode
Implements: StatementNode
Represents:
Base class for jump statements.
LABEL may be a NameNode or `AbsentTreeNode.instance` if absent.

ExprNode()
Parent type: TreeNode
Represents:
The base type for all expressions.

FieldAccessNode(NameNode node)
Parent type: ExprNode

Implements: NamedNode
Represents:
Base class for field accesses.

SingleExprNode(ExprNode expr)
Parent type: ExprNode
Represents:
An expression containing one embedded expression.

SingleOpNode(ExprNode expr)
Parent type: SingleExprNode
Represents:
An expression that operates on one embedded expression.

IncrDecrNode(ExprNode expr)
Parent type: SingleOpNode
Represents:
The common base type for pre/post increment/decrement expressions

UnaryArithNode(ExprNode expr)
Parent type: SingleOpNode
Represents:
The common base type for unary +/-

DoubleExprNode(ExprNode expr1, ExprNode expr2)
Parent type: ExprNode
Represents:
An expression containing two embedded expressions.

BinaryOpNode(ExprNode expr1, ExprNode expr2)
Parent type: DoubleExprNode
Represents:
An expression representing an operation between two embedded expressions.

BinaryArithNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for binary arithmetic operators

BinaryOpAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: DoubleExprNode
Represents:
An expression representing an operation between two embedded expressions, where the first expression takes the result value.

BinaryArithAssignNode(ExprNode expr1, ExprNode expr2)

Parent type: BinaryOpAssignNode
Represents:
Base class for assignment nodes `-= *= /= %=`.

ShiftNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for binary shift operators.

ShiftAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpAssignNode
Represents:
Base class for `<<= >>= >>>=`.

RelationNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for relations (i.e., `< > <= >=`).

EqualityNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for equality relations, `== !=`.

BitwiseNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for binary bitwise operations.

BitwiseAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpAssignNode
Represents:
Base class for `&= |= ^=`.

LogCondNode(ExprNode expr1, ExprNode expr2)
Parent type: BinaryOpNode
Represents:
Base class for `&& ||`.

LiteralNode(String ident)
Parent type: ExprNode
Represents:
Base class for literals.

OuterClassAccessNode(TypeNameNode type)
Parent type: ExprNode

Represents:
Base class for TYPE`.this` or TYPE`.super`.

Singletons

AbsentTreeNode.instance
Parent type: TreeNode
Represents:
A node that has been omitted in the tree.

BoolTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `boolean`.

CharTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `char`.

ByteTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `byte`.

ShortTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `short`.

IntTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `int`.

FloatTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `float`.

LongTypeNode.instance
Parent type: PrimitiveTypeNode
Represents:
The type `long`.

DoubleTypeNode.instance

Parent type: PrimitiveTypeNode
Represents:
The type `double`.

VoidTypeNode.instance
Parent type: TypeNode
Represents:
The type `void`.

NullTypeNode.instance
Parent type: ReferenceTypeNode
Represents:
A tree representing the “type” of `null`. This node does not appear in the AST.

Concrete classes

NameNode(TreeNode qualifier, String ident)
Parent type: TreeNode
Represents:
A name of the form QUALIFIER`.`IDENT, or, if QUALIFIER is absent, the name IDENT. When QUALIFIER is absent, this is called a **simple name**.

TypeNameNode(NameNode name)
Parent type: ReferenceTypeNode
Implements: NamedNode
Represents:
A type denoted by the NAME.

ArrayTypeNode(TypeNode elementType)
Parent type: ReferenceTypeNode
Represents:
The type ELEMENTTYPE `[]`.

ArrayInitTypeNode
Parent type: TypeNode
Represents:
The "type" of an array initializer. This node does not appear in the AST.

DeclaratorNode(int dims, TreeNode name, TreeNode initExpr)
Parent type: TreeNode
Represents:
A declarator of the form
NAME `[][]` ... = INITEXPR
where there are DIMS `[]`s.
INITEXPR is an expression; it may be `AbsentTreeNode.instance` if absent.
NAME is a simple name (see NameNode).

DeclaratorNodes must NOT appear in the AST produced by the parser. Instead, a declaration such as

```
int x, y[][], z = 2;
```

is represented by the same trees that would represent

```
int x;
int [][] y;
int z = 2;
```

(three FieldDeclNodes, LocalVarDeclNodes, etc.). The DeclaratorNode definition is here merely to aid the parser -- as a temporary data structure in which to hold a list of declarators.

CompileUnitNode(TreeNode package, List imports, List types)
Parent type: TreeNode
Implements: NamedNode
Represents:
The compilation unit
PACKAGE
IMPORTS
TYPES
PACKAGE is `AbsentTreeNode.instance` if absent, otherwise a NameNode.
IMPORTS is a list of ImportNode and ImportOnDemandNodes.
TYPES is a list of ClassDeclNodes and InterfaceDeclNodes.

ImportNode(NameNode name)
Parent type: TreeNode
Implements: NamedNode
Represents:
The single type import declaration
`import` NAME`;`

ImportOnDemandNode(NameNode name)
Parent type: TreeNode
Implements: NamedNode
Represents:
The declaration
`import` NAME`.*;`

ClassDeclNode(int modifiers, NameNode name, List interfaces, List members, TreeNode superClass)
Parent type: UserTypeDeclNode
Represents:
The class declaration
MODIFIERS `class` NAME `extends` SUPERCLASS `implements`
INTERFACES `{`
BODY
`}`

MODIFIERS is a combination (logical or) of values defined in Modifier.
NAME is a simple name (see NameNode).
SUPERCLASS is `AbsentTreeNode.instance` if the `extends` clause is absent; otherwise it is a TypeNameNode.
MEMBERS is a list of declarations of fields (see FieldDeclNode), methods (see MethodDeclNode), constructors (see ConstructorDeclNode), static initializers (see StaticInitNode), and instance initializers (see InstanceInitNode).

FieldDeclNode(int modifiers, TypeNode defType, NameNode name,
TreeNode initExpr)
Parent type: VarInitDeclNode
Represents:
A declaration of a field in a class or interface declaration of the form
MODIFIERS DEFTYPE NAME = INITEXPR;
or
MODIFIERS DEFTYPE NAME;
MODIFIERS is a combination (logical or) of values defined in Modifier.
NAME is a simple name (see NameNode).
INITEXPR is an ExprNode or `AbsentTreeNode.instance` if absent.

LocalVarDeclNode(int modifiers, TypeNode dtype, NameNode name,
TreeNode initExpr)
Parent type: VarInitDeclNode
Implements: StatementNode
Represents:
A declaration of a local variable in a method body of the form
MODIFIERS DTYPE NAME = INITEXPR;
or
MODIFIERS DTYPE NAME;
MODIFIERS is a combination (logical or) of values defined in Modifier.
NAME is a simple name (see NameNode).
INITEXPR is an ExprNode or `AbsentTreeNode.instance` if absent.

MethodDeclNode(int modifiers, NameNode name, List params, List throwsList,
TreeNode body, TypeNode returnType)
Parent type: InvokableDeclNode
Represents:
A method declaration of the form
MODIFIERS RETURNTYPE NAME `(`PARAMS`)` `throws` THROWSLIST
BODY
MODIFIERS is a combination (logical or) of values defined in Modifier.
RETURNTYPE is a type (any `[]`s that follow the formal parameter list must be incorporated into the RETURNTYPE).
NAME is a simple name (see NameNode).
PARAMS is a list of formal parameters (see ParameterNode).
THROWSLIST is a list of type names (see TypeNameNode).

BODY is a block (see BlockNode). If `abstract` is one of the MODIFIERS, then BODY may be `AbsentTreeNode.instance`.

ConstructorDeclNode(int modifiers, NameNode name, List params, List throwsList, ConstructorCallNode constructorCall, BlockNode body)
Parent type: InvokableNode
Represents:
A declaration of a constructor of the form

```
MODIFIERS IDENT ( PARAMS ) throws THROWS
{
  CONSTRUCTORCALL;
  BODY
}
```

MODIFIERS is a combination (logical or) of values defined in Modifier.
PARAMS is a list of formal parameters (see ParameterNode).
THROWS is a list of type names (see TypeNameNode).
CONSTRUCTORCALL is an explicit call to a constructor in this class or the superclass (see ThisConstructorCallNode and SuperConstructorCallNode). This may not be `AbsentTreeNode.instance`. If the source program does not contain an explicit constructor call, the parser must insert the equivalent of `super();` here.

ThisConstructorCallNode(List args)
Parent type: ConstructorCallNode
Implements: StatementNode
Represents:
An explict call to a constructor:

```
this(ARGS);
```

where ARGS is a list of argument expressions.
The DECL_KEY property should be set to the appropriate constructor.

SuperConstructorCallNode(List args)
Parent type: ConstructorCallNode
Implements: StatementNode
Represents:
An explict call to a constructor in the superclass:

```
super(ARGS);
```

where ARGS is a list of argument expressions. The DECL_KEY property should be set to the appropriate constructor.

StaticInitNode(BlockNode block)
Parent type: TreeNode
Represents:
A static initializer of the form

```
static BLOCK
```

InstanceInitNode(BlockNode block)

Parent type: TreeNode
Represents:
An instance initializer of the form

```
BLOCK
```

InterfaceDeclNode(int modifiers, NameNode name, List interfaces, List members)
Parent type: UserTypeDeclNode
Represents:
The interface declaration

```
MODIFIERS interface NAME extends INTERFACES {
  BODY
}
```

MODIFIERS is a combination (logical or) of values defined in Modifier.
NAME is a simple name (see NameNode).
INTERFACES is an empty list when there is no `extends` clause.
BODY is a list of declarations of fields (see FieldDeclNode), and methods without bodies (see MethodDeclNode).

ParameterNode(int modifiers, TypeNode defType, NameNode name)
Parent type: VarDeclNode
Represents:
A formal parameter:

```
DEFTYPE NAME
```

DEFTYPE is tree representing a type (see nodes with base type TypeNode). Any []s that follow NAME in the concrete syntax are folded into DTYPE. For example, if the text of a certain parameter is

```
int x[]
```

it is represented by the same tree as

```
int[] x
```

NAME is a simple name (see NameNode).

BlockNode(List stmts)
Parent type: TreeNode
Implements: StatementNode
Represents:
A statement of the form

```
{ STMTS }
```

STMTS is a list of statements.

EmptyStmtNode()
Parent type: TreeNode
Implements: StatementNode
Represents:
The empty statement (`;`)

LabeledStmtNode(NameNode name, StatementNode stmt)

Parent type: TreeNode
Implements: NamedNode
Represents:
A statement of the form
NAME: STMT

IfStmtNode(ExprNode condition, StatementNode thenPart, TreeNode elsePart)
Parent type: TreeNode
Represents:
A statement of the form
`if (`CONDITION`)` THENPART `else` ELSEPART
ELSEPART may be `AbsentTreeNode.instance` if absent.

SwitchNode(TreeNode expr, List switchBlocks)
Parent type: TreeNode
Implements: StatementNode
Represents:
A statement of the form
```
switch (EXPR) {
  SWITCHBLOCKS
}
```
SWITCHBLOCKS is a list of SwitchBranchNodes.

CaseNode(TreeNode expr)
Parent type: TreeNode
Represents:
A clause of the form
`case` EXPR`:`
in a branch of a switch statement. Represents a `default:` entry if EXPR is `AbsentTreeNode.instance`.

SwitchBranchNode(List cases, List stmts)
Parent type: TreeNode
Represents:
A portion of a switch statement of the form
CASES
STMTS
where CASES is a non-empty list of CaseNodes (representing case and default labels in a switch statement) and STMTS is a list of following statements.

LoopNode(TreeNode foreStmt, TreeNode test, TreeNode aftStmt)
Parent type: IterationNode
Represents:
A loop of the form
```
while (true) {
    FORESTMT
```

```
        if (!(TEST)) break;
        AFTSTMT
    }
```

By making FORESTMT be an EmptyStmtNode, this is a `while` loop.
By making AFTSTMT be an EmptyStmtNode, this is a `do-while` loop.

ExprStmtNode(ExprNode expr)
Parent type: TreeNode
Implements: StatementNode
Represents:
A statement of the form
`EXPR;`

ForNode(List init, ExprNode test, List update, StatementNode stmt)
Parent type: TreeNode
Implements: IterationNode
Represents:
A loop of the form
`for (`INIT`; `TEST`; `UPDATE`)` STMT
INIT is a list of declarations or statement expressions.
TEST is a `boolean` expression.
UPDATE is a list of statement expressions.

BreakNode(TreeNode label)
Parent type: JumpStmtNode
Represents:
The statement
`break` LABEL`;`
LABEL is `AbsentTreeNode.instance` when absent.

ContinueNode(TreeNode label)
Parent type: JumpStmtNode
Represents:
The statement
`continue` LABEL`;`
LABEL is `AbsentTreeNode.instance` when absent.

ReturnNode(TreeNode expr)
Parent type: TreeNode
Implements: StatementNode
Represents:
The statement
`return` EXPR`;`
EXPR is `AbsentTreeNode.instance` when absent.

ThrowNode(ExprNode expr)

Parent type: TreeNode
Implements: StatementNode
Represents:
The statement
`throw` EXPR`;`

SynchronizedNode(ExprNode expr, BlockNode block)
Parent type: TreeNode
Implements: StatementNode
Represents:
The statement
`synchronized (`EXPR`)` BLOCK

CatchNode(ParameterNode param, BlockNode block)
Parent type: TreeNode
Represents:
The clause
`catch (`PARAM`)` BLOCK

TryNode(BlockNode block, List catches, TreeNode finally)
Parent type: TreeNode
Represents:
The statement
`try`
BLOCK
CATCHES
`finally` FINLY
BLOCK and FINLY are blocks (see BlockNode). FINLY may be `AbsentTreeNode.instance` if absent.
CATCHES is a list of CatchNodes.

IntLitNode(String ident)
Parent type: LiteralNode
Represents:
A literal constant of type `int` whose external (written) representation is IDENT.

LongLitNode(String ident)
Parent type: LiteralNode
Represents:
A literal constant of type `long` whose external (written) representation is IDENT.

FloatLitNode(String ident)
Parent type: LiteralNode
Represents:

A literal constant of type `float` whose external (written) representation is IDENT.

DoubleLitNode(String ident)
Parent type: LiteralNode
Represents:
A literal constant of type `double` whose external (written) representation is IDENT.

BoolLitNode(String ident)
Parent type: LiteralNode
Represents:
The literal constant of type `boolean` whose external (written) representation is IDENT. IDENT is either `"false"` or `"true"`.

CharLitNode(String ident)
Parent type: LiteralNode
Represents:
A literal constant of type char whose external (written) representation is IDENT (a single character).

StringLitNode(String ident)
Parent type: LiteralNode
Represents:
A literal constant of type char whose external (written) representation is IDENT (a String of 0 or more characters).

NullPntrNode()
Parent type: ExprNode
Represents:
The null pointer, `null`.

ThisNode()
Parent type: ExprNode
Represents:
The expression `this`.

ArrayInitNode(List initializers)
Parent type: ExprNode
Represents:
An expression of the form
`{` INITIALIZERS `}`
used to initialize an array.
INITIALIZERS is a list of expressions (possibly including other ArrayInitNodes).

ArrayAccessNode(ExprNode array, ExprNode index)
Parent type: ExprNode
Represents:
The expression
ARRAY[INDEX]
where ARRAY and INDEX are expressions.

ObjectNode(NameNode name)
Parent type: ExprNode
Implements: NamedNode
Represents:
A reference to variable NAME (see NameNode) in an expression. This is used in all places where a name may be a reference to a variable (e.g., the expression `x+y` is represented by two ObjectNodes that contain NameNodes for `x` and `y`, respectively.) In cases like `x.y`, the context-free grammar cannot tell whether `x` denotes a class --so that x.y is simply a (static) variable, which should be represented as ObjectNode(N), where N is a NameNode for `x.y` -- of `x` denotes an object, so that `x.y` is a field access, which should be represented as ObjectFieldAccessNode(Y, X), where X is an ObjectNode for the simple variable `x` and Y is a NameNode for the simple name `y`. Therefore, the parser encodes `x.y` as a simple ObjectNode, and later parts of the compiler figure out whether to replace it with an ObjectFieldAccessNode.

ObjectFieldAccessNode(NameNode name, TreeNode object)
Parent type: FieldAccessNode
Represents:
A reference to a field of a general object of the form
OBJECT.NAME
See the documentation for ObjectNode.
NAME is a simple name (see NameNode).
OBJECT is an ExprNode.

SuperFieldAccessNode(NameNode name)
Parent type: FieldAccessNode
Represents:
A reference to a field or method in the superclass of the form
`super.`NAME
NAME is a simple name (see NameNode).

TypeFieldAccessNode(NameNode name, TypeNameNode fType)
Parent type: FieldAccessNode
Represents:
A reference to a static field or method of a class of the form
TYPE.NAME
NAME is a simple name (see NameNode).

MethodCallNode(ExprNode method, List args)

Parent type: ExprNode
Represents:
An expression of the form
METHOD (ARGS)
METHOD is an ObjectNode or field access (FieldAccessNode).
ARGS is a list of expressions.

AllocateNode(TypeNameNode dtype, List args)
Parent type: ExprNode
Represents:
An allocation of an instance of a class of the form
`new` DTYPE `(`ARGS`)`
or
ENCLOSINGINSTANCE `. new` DTYPE `(`ARGS`).`
ARGS is a list of expressions.
ENCLOSINGINSTANCE is a ExprNode or `AbsentTreeNode.instance` if absent.
The DECL_KEY property identifies the constructor to be used.

AllocateAnonymousClassNode(TypeNameNode superType, List superArgs, List members, TreeNode enclosingInstance)
Parent type: ExprNode
Represents:
An allocation of an anonymous class of the form
`new` SUPERTYPE `(`SUPERARGS`) {`
MEMBERS
`}`
or
ENCLOSINGINSTANCE `. new` SUPERTYPE `(`SUPERARGS`) {`
MEMBERS
`}`
ARGS is a list of expressions.
MEMBERS is a list of declarations of fields (see FieldDeclNode), methods (see MethodDeclNode), static initializers (see StaticInitNode), and instance initializers (see InstanceInitNode).
ENCLOSINGINSTANCE is a ExprNode or `AbsentTreeNode.instance` if absent.
The DECL_KEY property identifies the constructor to be used.

AllocateArrayNode(TypeNode dtype, List dimExprs, int dims, TreeNode initExpr)
Parent type: ExprNode
Represents:
An expression of the form
`new` DTYPE DIMEXPRS `[][]` ... INITEXPR
where there are DIMS `[]`s.

DIMEXPRS is a list of expressions (it looks like `[EXPR0][EXPR1]`... in an actual program).
INITEXPR is an ArrayInitExpr or `AbsentTreeNode.instance` if absent.

PostIncrNode(ExprNode expr)
Parent type: IncrDecrNode
Represents:
The expression
EXPR ++

PostDecrNode(ExprNode expr)
Parent type: IncrDecrNode
Represents:
The expression
EXPR - -

UnaryPlusNode(ExprNode expr)
Parent type: UnaryArithNode
Represents:
The expression
+ EXPR

UnaryMinusNode(ExprNode expr)
Parent type: UnaryArithNode
Represents:
The expression
- EXPR

PreIncrNode(ExprNode expr)
Parent type: IncrDecrNode
Represents:
The expression
++ EXPR

PreDecrNode(ExprNode expr)
Parent type: IncrDecrNode
Represents:
The expression
- - EXPR

ComplementNode(ExprNode expr)
Parent type: SingleOpNode
Represents:
The expression
~ EXPR

NotNode(ExprNode expr)
  Parent type: SingleOpNode
 Represents:
  The expression
      ! EXPR

CastNode(TypeNode dtype, ExprNode expr)
  Parent type: ExprNode
 Represents:
  The expression
      (DTYPE) EXPR

MultNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithNode
 Represents:
  The expression
      EXPR1 * EXPR2

DivNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithNode
 Represents:
  The expression
      EXPR1 / EXPR2

RemNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithNode
 Represents:
  The expression
      EXPR1 % EXPR2

PlusNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithNode
 Represents:
  The expression
      EXPR1 + EXPR2

MinusNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithNode
 Represents:
  The expression
      EXPR1 - EXPR2

LeftShiftLogNode(ExprNode expr1, ExprNode expr2)
  Parent type: ShiftNode
 Represents:

The expression
EXPR1 << EXPR2

RightShiftLogNode(ExprNode expr1, ExprNode expr2)
Parent type: ShiftNode
Represents:
The expression
EXPR1 >>> EXPR2

RightShiftArithNode(ExprNode expr1, ExprNode expr2)
Parent type: ShiftNode
Represents:
The expression
EXPR1 >> EXPR2

LTNode(ExprNode expr1, ExprNode expr2)
Parent type: RelationNode
Represents:
The expression
EXPR1 < EXPR2

GTNode(ExprNode expr1, ExprNode expr2)
Parent type: RelationNode
Represents:
The expression
EXPR1 > EXPR2

LENode(ExprNode expr1, ExprNode expr2)
Parent type: RelationNode
Represents:
The expression
EXPR1 <= EXPR2

GENode(ExprNode expr1, ExprNode expr2)
Parent type: RelationNode
Represents:
The expression
EXPR1 >= EXPR2

InstanceOfNode(ExprNode expr, TreeNode dtype)
Parent type: ExprNode
Represents:
The expression
EXPR `instanceof` DTYPE

EQNode(ExprNode expr1, ExprNode expr2)

Parent type: EqualityNode
Represents:
The expression
EXPR1 == EXPR2

NENode(ExprNode expr1, ExprNode expr2)
Parent type: EqualityNode
Represents:
The expression
EXPR1 != EXPR2

BitAndNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseNode
Represents:
The expression
EXPR1 & EXPR2

BitOrNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseNode
Represents:
The expression
EXPR1 | EXPR2

BitXorNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseNode
Represents:
The expression
EXPR1 ^ EXPR2

CandNode(ExprNode expr1, ExprNode expr2)
Parent type: LogCondNode
Represents:
The expression
EXPR1 && EXPR2

CorNode(ExprNode expr1, ExprNode expr2)
Parent type: LogCondNode
Represents:
The expression
EXPR1 || EXPR2

IfExprNode(ExprNode expr1, ExprNode expr2, ExprNode expr3)
Parent type: ExprNode
Represents:
The expression
EXPR1 ? EXPR2 : EXPR3

AssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: DoubleExprNode
 Represents:
  The expression
      EXPR1 = EXPR2

MultAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithAssignNode
 Represents:
  The expression
      EXPR1 *= EXPR2

DivAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithAssignNode

 Represents:
  The expression
      EXPR1 /= EXPR2

RemAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithAssignNode
 Represents:
  The expression
      EXPR1 %= EXPR2

PlusAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithExprNode
 Represents:
  The expression
      EXPR1 += EXPR2

MinusAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: BinaryArithAssignNode
 Represents:
  The expression
      EXPR1 -= EXPR2

LeftShiftLogAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: ShiftAssignNode
 Represents:
  The expression
      EXPR1 <<= EXPR2

RightShiftLogAssignNode(ExprNode expr1, ExprNode expr2)
  Parent type: ShiftAssignNode

Represents:
The expression
EXPR1 >>>= EXPR2

RightShiftArithAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: ShiftAssignNode
Represents:
The expression
EXPR1 >>= EXPR2

BitAndAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseAssignNode
Represents:
The expression
EXPR1 &= EXPR2

BitXorAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseAssignNode
Represents:
The expression
EXPR1 ^= EXPR2

BitOrAssignNode(ExprNode expr1, ExprNode expr2)
Parent type: BitwiseAssignNode
Represents:
The expression
EXPR1 |= EXPR2