A Code Generation Framework for Java Component-Based Designs[1]

Jeff Tsay
BDTI
2107 Dwight Way
Berkeley, CA 94704
(1) 510-665-1600
ctsay@cs.berkeley.edu

Christopher Hylands
EECS, UC Berkeley
Cory Hall
Berkeley, CA 94720
(1) 510-643-9841
cxh@eecs.berkeley.edu

Edward A. Lee
EECS, UC Berkeley
Cory Hall
Berkeley, CA 94720
(1) 510-642-0455
eal@eecs.berkeley.edu

ABSTRACT

In this paper, we describe a software architecture supporting code generation from within Ptolemy II. Ptolemy II is a component-based design tool intended for embedded and real-time system design. The infrastructure described here provides a platform for experimentation with synthesis of embedded software or hardware designs from high-level, Java-based specifications. A specification consists of a set of interconnected actors whose functionality is defined in Java. The provided infrastructure parses the existing Java code for the actors and presents a simple API, using the Visitor pattern, for transforming the abstract syntax tree and writing code transformers and back-end code generators. A code transformer is described that performs generic optimizations, such as specialization of polymorphic data types, and domain-specific optimizations, such as static buffer allocation for communication between dataflow actors. The back end resynthesizes the transformed Java code. We describe a simple example where the generated Java code runs on a PalmOS handheld computer, and a more elaborate example where significant performance improvements are achieved by transformations.

Keywords. Autocoding, Compilers, Embedded Software, Generators, Java.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES '00, November 17-19, 2000, San Jose, CA.
Copyright 2000 ACM 1-58113-338-3/00/0011…$5.00.

1. INTRODUCTION

Component-based design systems, such as the popular block-diagram languages SPW and Simulink, face a difficult tradeoff between programmer convenience and efficiency of execution. Since they are often targeted to embedded software design, efficiency is essential. The generator approach offers a solution. Programmer convenience is emphasized during the development phase, and then designs are translated (by "generators" or "auto coders") to optimized implementations.

There are several approaches to building generators. The most obvious one is to adapt traditional compilers to component-based designs, and build in optimizations suited to real-time embedded systems (see for example [4]). In this approach, the component architecture is that of an object-oriented language, such as C++. Many designers, however, prefer concurrent, actor-based modeling such as that used in block diagram languages, over OO architectures. An alternative generator approach that matches these languages well is the library-based approach. In one library-based approach, used in a number of commercial and non-commercial block-diagram programming environments, each component is responsible for its own representation in the target language. Ptolemy Classic, the predecessor of Ptolemy II, is such a block-diagram environment [5]. It is capable of synthesizing C, C++, VHDL, and assembly code for DSPs. But the block library becomes impossibly difficult to maintain because the library maintainer has to write code generators for each block for each target language.

A third approach is to write libraries in a common base language, such as C, and build generators that can manipulate both the C and the concurrent, actor-based component architecture. The MathWorks Real-Time Workshop is such a generator. It generates C code from a block diagram specified in Simulink. The code generation process consists of two steps: generation of "C MEX S-functions" from a Simulink block diagram, and

[1] This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, Hughes Space and Communications, Lockheed-Martin (Sanders division), and Philips.

generation of C code from these functions. For each block in the block diagram, the behavior of the code generator can be customized for efficiency, and many of the built-in blocks are dealt with specially. Such behavioral changes are described in another language and compiled by the "Target Language Compiler". By combining automatically generated code based on

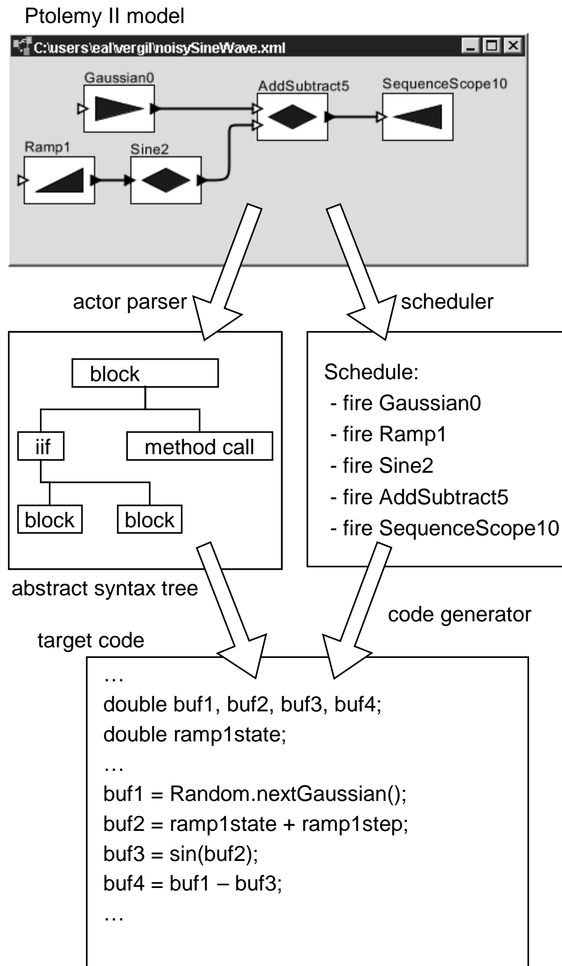

Figure 1. Outline of generator strategy

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.

Our approach [6] is similar to that of the Real-Time Workshop, but with an emphasis on building a framework supporting a plug-in architecture for writing optimized code generators for a variety of domains. Actors in a library are written in Java. The Java code is parsed, and an API is defined for manipulating the abstract syntax tree (AST). We show how manipulations of the AST can take advantage of type resolution, inter-actor communication semantics, and other specialized features. An outline of the approach is shown in Figure 1. The left path in this figure shows the parsing of actors in the component library. The right path shows using the inter-component semantic model to generate schedules (and other domain-specific optimizations).

2. PTOLEMY II

The Ptolemy II software package [2] provides a framework for modeling and 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 is 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. Another domain is the synchronous dataflow (SDF) domain, in which all actors may be executed sequentially according to a static schedule. A third domain, discrete events (DE), has semantics similar to that used in hardware description languages such as VHDL and Verilog. A fourth domain, finite state machines (FSM), provides sequential rather than concurrent semantics.

In addition to ports, actors may have parameters, which may be configured at run-time for more flexibility. Parameters can be queried for and set with token values. Parameters are (by convention) public fields of actors, so they can be queried and set from outside of the actor class. Since both ports and parameters provide inputs to actors, it is not always clear which to use. We tend to use parameters to provide configuration information, which changes relatively infrequently, and ports to provide run-time data, which changes relatively frequently. This distinction becomes important for code generation, since many optimizations become possible if one assumes that a particular parameter value will not change during execution.

An executable system in Ptolemy II 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 action methods of actors:

- preinitialize(), invoked exactly once before execution begins;
- initialize(), invoked exactly once after preinitialize();
- prefire(), invoked once per iteration;
- fire(), invoked once or more per iteration;
- postfire(), invoked once per iteration, after the last fire(); and
- wrapup(), invoked once when (and if) execution ends.

During the execution of most of these, the actor may read a token from a channel of a port by calling port.get(*c*), where *c* is the channel number. It may write a token to a channel of a port by calling port.send(*c*, *t*), where *t* is a token. Finally, it may write to all channels of a port by calling port.broadcast(*t*). Typically, the majority of the real work is performed inside the fire() method.

Ptolemy II is rich enough that systems using the different domains still use the same basic kernel. This richness allows for heterogeneous modeling, in which two or more domains may be used simultaneously in the same system, and domain-polymorphic actors, which are actors that execute in more than one domain. In addition, because data is encapsulated in tokens, actors may also be data 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 data and domain polymorphism, consider the AddSubtract actor. The actor executes the following steps in its fire() method:

- Read token(s) from the *plus* port, and “add” them to the initially “zero” result.
- Read token(s) from the *minus* port, and “subtract” them from the result.
- 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. Moreover, the add and subtract operations are performed by invoking polymorphic method calls on the Token base class, so the same code be can used to add integers, floating-point numbers, matrices, or even strings. Thus, the actor is also data polymorphic.

Ptolemy II provides a sophisticated type system [8] that statically resolves data types to the most specific type that meets all specified constraints. Thus, if an AddSubtract actor is used in a context where it will always be supplied with double precision floating-point numbers, then its ports will resolve to type *double*.

3. CODE GENERATION STRATEGY

Actors in the actor library are written to maximize their reusability, hence their domain and data polymorphism. When generating code for a particular application, we can make use of static information to considerably reduce the cost of implementing the actor. In particular,

- Knowing the domain of the actor, its read and write operations through ports can be replaced with optimized code, for example to read and write directly from/to statically allocated buffers.
- Knowing the resolved data types of the ports, data polymorphic operations such as arithmetic performed via method calls can be replaced with resolved operations on the appropriate types.
- Knowing the parameter values of the actor, references to parameters can be replaced by constants, and dead code that cannot execute with the given parameters can be eliminated.
- Dependencies on large packages, such as Ptolemy II packages and Java class libraries, can be eliminated by only considering code for the methods that are actually used.

3.1 Outline of Generator Sequence

The code generator proceeds through the following steps for each actor in a system:

1. The source code is parsed and static semantic analysis is performed, yielding a decorated abstract syntax tree (AST).
2. 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.
3. The resulting AST is transformed into an intermediate AST, 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 ports are transformed in a domain-specific way.
4. The AST is reverted back to an ordinary Java AST by adding conversions that widen types and method calls that implement matrix addition, etc.
5. Finally, target source code is regenerated from the transformed AST.

These steps are illustrated in Figure 2, where they are shown as transformations on the source code rather than on the AST.

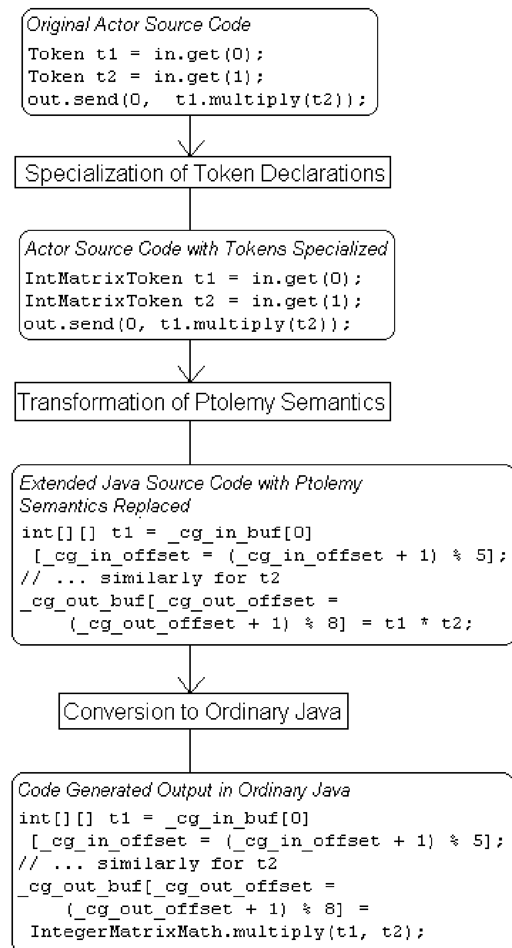

Figure 2. Sequence of transformations, performed on the AST.

3.2 Compiler Tools

The Java compiler in Ptolemy II is based on the source code for the Titanium compiler [8]. The source code for the Titanium compiler itself is written in C++, which we translated into Java. In addition, we made a major design change. 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. Besides type resolution and back-end code generation, another example of a coherent operation would be strength reduction, in which nodes representing multiplication by two would be replaced by nodes representing the addition of a variable to itself. The approach taken in the Titanium compiler has the following disadvantages:

- The code to do one coherent operation is spread over all node classes, making the code difficult to maintain and debug.
- Additional operations can only be added by modifying the source code for each node.

The latter problem is the most critical. Recall that our objective is to build an infrastructure for code generation. Other programmers need to be able to add their own facilities. It would be unreasonable to require them to modify the node classes in order to add their own (possibly highly specialized) facilities for manipulating the AST.

We solve this problem with the Visitor pattern [3], shown in Figure 3. The source code for each node contains only a few access methods, and an _acceptHere() method. The _acceptHere() method takes a visitor as an argument, and passes the node to the appropriate method of the visitor. Each type of visitor is responsible for one coherent operation to be performed on an AST, and has one method for each of the types of nodes that may appear in the AST.

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. The loss in node class code reuse is offset by the gain in visitor class code reuse; a visitor class can inherit code from other visitor classes. Thus, an operation may be specialized.

TreeNode, the base class for all nodes that appear in an AST, is designed to interact with visitors. Each node contains a child list, which is a list whose members may be instances of TreeNode or other child lists. Therefore, in subclasses of TreeNode, it is not necessary to declare explicit fields for child nodes or lists of nodes. The accept() method takes a visitor as an argument, performs automatic visitation of the child list if desired, and eventually calls _acceptHere(). Properties (or attributes) may be put and retrieved from each instance of TreeNode. 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 visitor. Later, a property may be retrieved by another visitor.

Since our major objective is to create an infrastructure for code generation, we created a visualization tool, shown in Figure 4, that displays an AST. This tool is particularly useful when writing transformations on an AST, so that effects of transformations are immediately visible.

3.3 Java Compiler Front End

We used JLex[2] to generate a lexical analyzer for Java, and BYACC/J[3] to generate a parser. The specification files for both the lexical analyzer and the parser were based on those of the Titanium compiler, but we added additional rules for Java 1.2 and removed Titanium-specific extensions.

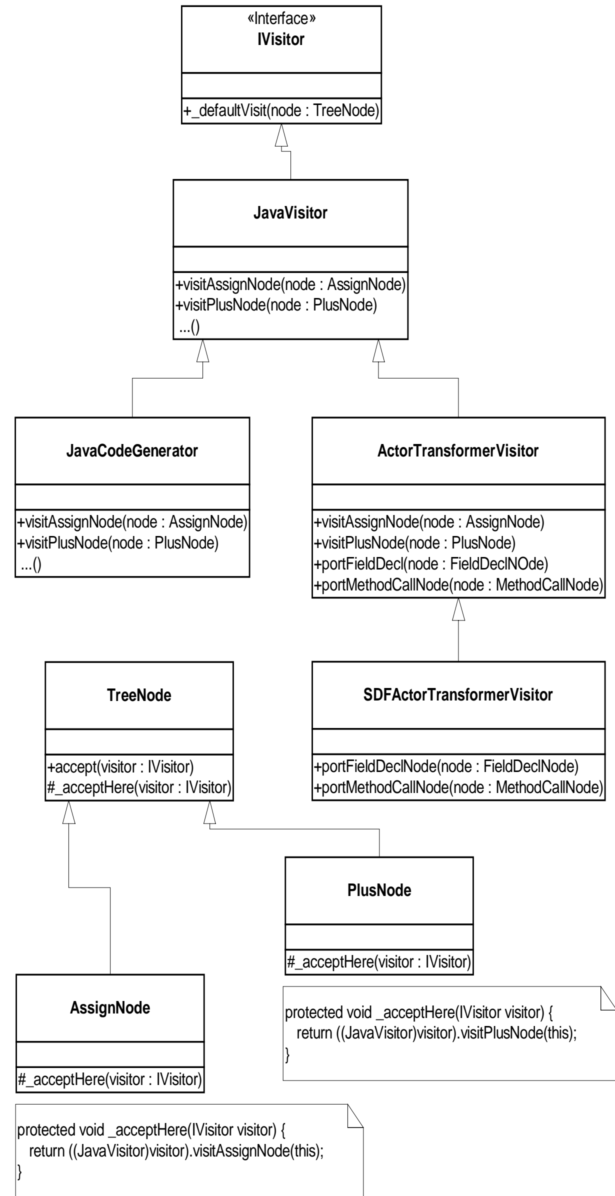

Figure 3. The Visitor pattern in UML.

[2] http://www.cs.princeton.edu/~appel/modern/java/JLex/

[3] http://www.lincom-asg.com/~rjamison/byacc/

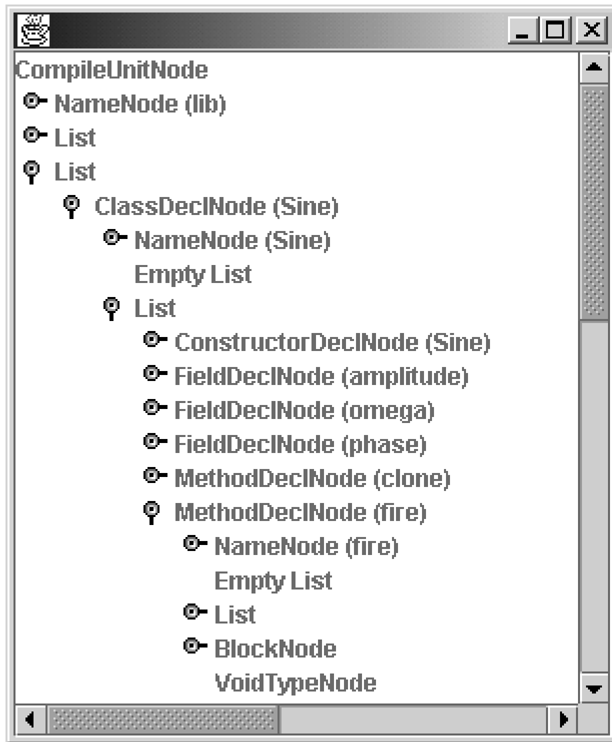

Figure 4. A visualization tool for an abstract syntax tree.

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. Visitors are used extensively in this step.

Our current implementation of static semantic analysis requires access to the source code of Java class libraries that are used. However, we are redesigning the code to use reflection instead.

A feature of the Java compiler in Ptolemy II is that rules for determining types can be changed. This is accomplished with 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 of 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(), 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.

3.4 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. Other target languages, such as C or VHDL, can also be generated, in principle, although we have currently implemented only a Java back end. The back end again uses a visitor to convert the AST into text representing source code written in Java.

3.5 Code Generation Framework Structure

The Ptolemy II code generator 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 reading and writing data to a port. The code is designed so that the domain-independent parts are clearly separated from the domain-specific parts. To implement a code generator for a specific domain, one simply writes an AST transformer visitor that extends the domain-independent AST transformer, overloading the appropriate methods that handle transformations of I/O method calls, and a subclass of an Abstract Factory [3] that creates the domain-specific transformer. Thus, a domain-specific transformer inherits the code that does domain-independent transformation, and can be used alone.

The code generation process as described in section 3.1 is independent of the domain until step 3, at which time a domain-specific transformer is created by the Abstract Factory and proceeds to transform the AST. After step 3, the code generation process is again domain-independent.

3.6 Domain-Independent Code Generation

The domain-independent parts of the code generation process perform three main kinds of transformation. The first kind is the replacement of declarations of tokens with declarations of the appropriate resolved data types and the replacement of abstract operations on tokens with the appropriate resolved operations, based on the types of the operand(s). In order to do these transformations, token variables must be analyzed to determine the most specific concrete type that can represent them, which occurs in step 2 of the process. Step 2 makes use of the resolved types of the ports and parameters of the actor, and uses the Ptolemy II type lattice to solve inequalities in which the variables are types of token variables. Step 2 then replaces the declared type of token variables with the most specific token type possible. For instance, a variable declared with type Token might be declared with type DoubleMatrixToken after step 2. In step 3, token types are mapped to the types that they encapsulate. Continuing the above example, the variable declared with type DoubleMatrixToken would be then declared with type *double[][]*. AST nodes that represent method calls performing tokens operations are temporarily replaced by expression nodes that may or may not valid in ordinary Java. For example, the expression `a.add(b)` is replaced with the expression `a + b`, where `a` and `b` were DoubleMatrixTokens before step 3, and instances of *double[][]* after step 3. Using the final types of the operands, step

4 transforms normally illegal expressions such as `a + b` into method calls such as `DoubleMatrixMath.add(a, b)`, which is a call to a static method in the Ptolemy `math` package.

The second kind of domain-independent transformation is the replacement of parameter queries with references to parameter variables. When a code generated actor is instantiated, its parameter variables are set to whatever data is present in the corresponding parameters during simulation, since parameters values are assumed constant.

The third kind of domain-independent transformation is the elimination of code that is unnecessary for the code generated version of the actor. Code elimination is possible just by removing calls to the Ptolemy kernel that configure the system (declaring the types of ports, etc.). This has the effect of eliminating dependencies on the Ptolemy classes used for simulation.

3.7 Domain-Specific Code Generation

A domain-specific code generator uses semantic properties of the actor interactions to optimize the code. We have built only one of these at this point, for the synchronous dataflow (SDF) domain. In SDF, actors run sequentially and communicate via fixed-sized FIFO buffers. The order in which actors execute is determined by a statically computed schedule. 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, and considerable technology exists for choosing schedules [1]. So far, we have only used a simple, non-optimizing scheduler.

The first step of the SDF code generator is to determine the size of the 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 $p + at$, where p = the number of initial tokens, a = the number of appearances in the schedule, and t = the number of tokens produced per appearance. The buffer is used circularly, so increasing offsets into the buffer must be taken modulo the buffer size.

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

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

The name `_cg_out_2` is generated automatically. Variables and code are also automatically generated to handle indexing into this buffer. These variables are used as follows. If an actor contains the statement

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

this will be transformed to

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

The transformation can get a bit more complicated when the indices are not constants, but are rather computed values, but the transformed code still executes much more efficiently than the domain and data polymorphic call to the send() method.

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 bit-wise AND operations. If the buffer size is one, then 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. in an order determined by Ptolemy semantics and the scheduler.

The output code generated system is close to a system optimized by a human being; all calls the Ptolemy kernel are replaced by efficient code. Thus, by using an ordinary Java to C converter, the code generated system could be transformed into C code for compilation for an embedded processor.

4. EXAMPLE

We performed SDF code generation on a model does the transmission and detection of discrete-time, orthogonal signals, a block diagram of which is shown in Figure 5. One of two orthogonal signals is selected based on the input bit, which is generated by a Bernoulli process. The signal is then corrupted by additive white Gaussian noise before reception. The received signal is correlated with the two original signals by performing the dot product operation. The waveform with the maximum correlation corresponds to the decision made on which waveform was transmitted.

All dependencies on the Ptolemy `actor`, `kernel`, `data`, and `domains.sdf.kernel` packages are eliminated by the code transformer. However, additional dependencies on the Ptolemy `math` package are 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. The just-in-time compiler makes measuring execution time fairly tricky[4], but we found that the code generated version ran anywhere from 2.5 to 10 times faster than the original version. We also found that size of the `jar` file necessary to include all of the runtime Ptolemy II classes was roughly one-tenth the size in the code generated version. In general, the performance improvement of a code generated system over the corresponding simulated system will depend heavily on the granularity of the actors.

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

1. Input from multiple channels;
2. Output to multiple destinations;

[4] "There are three kinds of lies: lies, damned lies, and statistics." Mark Twain attributes this to Disraeli.

3. Multirate dataflow;
4. Data polymorphic addition of DoubleMatrixTokens;
5. Unconnected ports; and
6. Use of a mix of SDF-specific and domain-polymorphic actors.

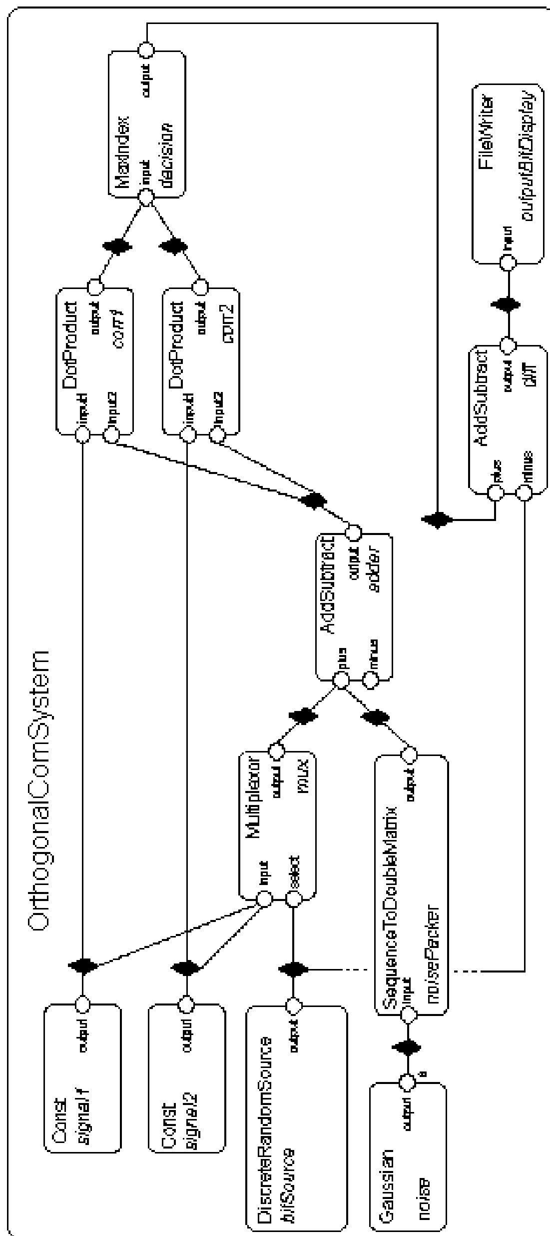

Figure 5. Example Ptolemy II model.

5. GENERATING SPECIALIZED CODE

A major objective is to be able to translate high level specifications into embedded software. To demonstrate the potential here, we selected the PalmOS as a target. To keep our task simple, we use Java on the Palm Pilot, running a small JVM.

In June 1999, Sun Microsystems released a Java Virtual Machine that ran under PalmOS. The first release of this product was called the KVM, where the K stood for kilobytes, indicating that the virtual machine would run in a very small memory footprint (50-80k of object code).[5] In May 2000, Sun released the Java 2 Micro Edition (J2ME), which uses the KVM and Connected, Limited Device Configuration (CLDC) technology to provide a Java environment under PalmOS. The CLDC implementation for PalmOS is fairly limited in many ways. Two key limitations are that floating point data types are not supported, and it is not possible to access native code using JNI. The Palm OS Emulator (POSE), available from the Palm website, offers a convenient code development platform, so we used it primarily, and only occasionally ran code on the actual hardware (which was a Handspring Visor).

The limitation of not having floating point data types is severe, and hence we could not run anything as interesting as the model in Figure 5 on the PalmOS. Nonetheless, we were able to run a simple model consisting of a Ramp actor connected to a FileWriter actor that displayed steadily increasing integers. This example demonstrates that models that use integer arithmetic can be translated into very small images that operate independently of the Ptolemy infrastructure.

We also used Waba, an open source vm implementation for the Palm. Waba has the advantage that it supports floats, but Waba does not support exceptions and has limited i/o capability. We were able to run a simple system that used floats by editing the generated code by hand. We believe that it would be fairly straight forward to generate code for Waba directly.

6. APPLICATIONS

There are a number of applications for the infrastructure described in this paper. First, it can be used simply to transform one Java program into a leaner realization (faster and smaller). Second, it can be used to transform a Java program into an embedded version, for execution on a much smaller virtual machine. We have demonstrated both of these applications with simple examples. We can also speculate on a number of other applications.

For example, a full Java compiler could be completed by creating new back ends as visitors of a Java AST. Also, a number of utility programs might be implemented using visitors. An example of such a program, which we have implemented, 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 a natural solution. It would also be possible to implement a Java to C converter using visitors, although a number of programs that do this conversion already exist. 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 operators between special types like matrices. Extended Java was created only 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

[5] http://java.sun.com/products/cldc/ds/

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 scenarios.

The code generator for SDF might be modified to generate code executable in parallel processing environments, if a suitable parallel scheduler were used instead of the existing scheduler. The modifications to the code generator would entail generating a main() method that executes actors in parallel, and modifying certain buffer accesses to block, waiting for other execution paths to complete.

Our intention is to release the Ptolemy II code generation facility with the next release of Ptolemy II, due out in 2000Q4. Please consult the Ptolemy II website at:

http://ptolemy.eecs.berkeley.edu/ptolemyII

7. REFERENCES

[1] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee. "Synthesis of Embedded Software from Synchronous Dataflow Specifications," Journal of VLSI Signal Processing 21, 151-166, 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/synthesis/)

[2] 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. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/HMAD)

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, ISBN 0-201-63361-2, October 1994.

[4] S. Malik, S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, G. Araujo, A. Sudarsanam, V. Ziviojnovic, and H. Meyr, "Code Generation and Optimization Techniques for Embedded Digital Signal Processors," in *Hardware Software Codesign*, Kluwer Academic Publishers, NATO-ASI Series, 1996, G. DeMicheli and M. Sami, Editors.

[5] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck, "Software Synthesis for DSP Using Ptolemy," Journal on VLSI Signal Processing, vol. 9, no. 1, pp. 7-21, Jan., 1995. (http://ptolemy.eecs.berkeley.edu/publications/papers/95/jvsp_codegen)

[6] Jeff Tsay, "A Code Generation Framework for Ptolemy II," ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/00/codegen/)

[7] Y. Xiong and E. A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. LNCS 1785, Springer-Verlag.

[8] 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,CA.