A Code Generation Framework for Actor-Oriented Models with Partial Evaluation

Gang Zhou, Man-Kit Leung, and Edward A. Lee

University of California, Berkeley
{zgang,jleung,eal}@eecs.berkeley.edu

Abstract. Embedded software requires concurrency formalisms other than threads and mutexes used in traditional programming languages like C. Actor-oriented design presents a high level abstraction for composing concurrent components. However, high level abstraction often introduces overhead and results in slower system. We address the problem of generating efficient implementation for the systems with such a high level description. We use partial evaluation as an optimized compilation technique for actor-oriented models. We use a helper-based mechanism, which results in flexible and extensible code generation framework. The end result is that the benefit offered by high level abstraction comes with (almost) no performance penalty. The code generation framework has been released in open source form as part of Ptolemy II 6.0.1.

1 Introduction

Embedded software has been traditionally written with assembly language to maximize efficiency and predictability. Programming languages like C are used to improve productivity and portability. These imperative programming languages essentially abstract how a Von Neumann computer operates in a sequential manner. They are good matches for general-purpose software applications, which are essentially a series of data transformations. However, in the embedded world, the computing system constantly engages the physical system. Thus the physical system becomes an integral part of the design and the software must operate concurrently with the physical system. The basic techniques for doing concurrent programming on top of traditional programming languages like C use threads, complemented with synchronization mechanisms like semaphores and mutual exclusion locks. These methods are at best retrofits to the original fundamentally sequential formalism. Therefore they are difficult to reason about and guarantee correctness [1]. In fact, according to a survey [2] conducted by the Microsoft Windows Driver Foundation team, the top reason for driver crashes is concurrency and race conditions. This is not acceptable for embedded applications that are real-time and often safety-critical. We need alternative concurrency formalisms to match the abstractions with the embedded applications.

A number of design frameworks have emerged over the years that offer different concurrency models for the applications they support. For example, StreamIt [3] has a dataflow formalism nicely matched to streaming media applications.

Y.-H. Lee et al. (Eds.): ICESS 2007, LNCS 4523, pp. 786–799, 2007.

Simulink [4] has roots in control system modeling, and time is part of its formal semantics. All these frameworks have formal concurrent models of computation (MoCs) that match their application spaces. They often use block diagram based design environments and the design usually starts with assembling pre-existing components in the library. Such design style has been called domain specific design, model based design or component based design, each emphasizing different aspects of the design. Many of them employ an actor-oriented approach, where actor is an encapsulation of parameterized actions performed on input data and produce output data. Input and output data are communicated through well-defined ports. Ports and parameters form the interface of an actor. Actor-oriented design hides the state of each actor and makes it inaccessible from other actors. The emphasis of data flow over control flow leads to conceptually concurrent execution of actors. Threads and mutexes become implementation mechanism instead of part of programming model.

A good programming model with abstraction properties that match the application space only solves half of the problem. For it to succeed, it is imperative that an efficient implementation be derived from a design described in the programming model. In component based design (we use the terms "actor" and "component" interchangeably in this paper), modular components make systems more flexible and extensible. Different compositions of the same components can implement different functionality. However, component designs are often slower than custom-built code. The cost of inter-component communication through the component interface introduces overhead, and generic components are highly parameterized for the reusability and thus less efficient.

To regain the efficiency for the implementation, the users could write big monolithic components to reduce inter-component communication, and write highly specialized components rather than general ones. Partial evaluation [5] provides an alternative mechanism that automates the whole process. Partial evaluation techniques have recently begun to be used in the embedded world, e.g, see [6]. We use partial evaluation for optimized code generation, transforming an actor-oriented model into target code while preserving the model's semantics. However, compared with traditional compiler optimization, our partial evaluation works at the component level and heavily leverages domain-specific knowledge. Through model analysis, the tool can discover data types, buffer sizes, parameter values, model structures and model execution schedules, and then partially (pre)evaluate all the known information to reach an efficient implementation. The end result is that the benefit offered by the high level abstraction comes with (almost) no performance penalty.

1.1 Related Work

There have been a few design frameworks with code generation functionality. Simulink with Real-Time Workshop (RTW), from the Mathworks, is probably in the most widespread use as a commercial product [4]. It can automatically generate C code from a Simulink model and is quite innovative in leveraging an underlying preemptive priority-driven multitasking operating system to deliver

real-time behavior based on rate-monotonic scheduling. However, like most design frameworks, Simulink defines a fixed MoC: continuous time is the underlying MoC for Simulink, with discrete time treated as a special case (discrete time signal is piecewise-constant continuous time signal in Simulink). It can be integrated with another Mathworks product called Stateflow, used to describe complex logic for event-driven systems. The platform we are working on, Ptolemy II, is a software lab for experimenting with multiple concurrency formalisms for embedded system design. It does not have a built-in MoC. The code generation framework built on Ptolemy II is flexible and extensible. It is capable of generating code for multiple MoCs. In particular, we are most interested in generating code for those MoCs for which schedulability is decidable.

Partial evaluation has been in use for many years [5]. The basic idea is that given a program and part of this program's input data, a partial evaluator can execute the given program as far as possible producing a residual program that will perform the rest of computation when the rest of the input data is supplied. It usually involves a binding-time analysis phase to determine the static parts and the dynamic parts of a program, followed by an evaluation phase. The derived program is usually much more efficient by removing computation overhead resulting from the static parts. Partial evaluation has been applied in a variety of programming languages including functional languages [7], logic languages [8], imperative languages like C [9] and object-oriented languages [10]. Its use in embedded software has been more recent. Click is a component framework for PC router construction [6]. It builds extensible routers from modular components which are fine-grained packet processing modules called elements. Partial evaluations are applied at the level of components using optimization tools called click-fastclassifier, click-devirtualize, click-xform, click-undead, etc. For example, classifiers are generic elements for classifying packets based on a decision tree built from textual specifications. The click-fastclassifier tool would generate new source code for a classifier based on the specific decision tree and replace the generic element with this more specific element. The click-devirtualize tool addresses virtual function call overhead. It changes packet-transfer virtual function calls into conventional function calls by finding the downstream component and explicitly calling the method on that component. Again this involves transforming the source code so that method binding can be done in the compile time. The Koala component model for consumer electronics software is another example of applying partial evaluation for generating more efficient implementation [11]. Compared with previous examples, Ptolemy II does not focus on specific applications. Its emphasis is on choosing appropriate concurrent MoCs for embedded system design and generating efficient code for the chosen MoCs.

There has been previous work on code generation for Ptolemy II [12]. The approach there involves transformation of the existing source code (i.e. Java code) in each actor of a system, which results in simplified and hence more efficient Java code. Then a generic Java-to-C converter is used to produce compilable C code. The generated code is not efficient enough to be useful for embedded application. However, the techniques developed there such as specializing token

declarations, assigning static offsets to input and output token buffers, static scheduling of SDF actors are useful and can be equally applied in our context.

1.2 Overview of the Code Generation Framework

Ptolemy II is a graphical software system for modeling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II focuses on assembly of concurrent components with well-defined MoCs that govern the interaction between components. Many features in Ptolemy II contribute to the ease of its use as a rapid prototyping environment. For example, domain polymorphism allows one to use the same component in multiple MoCs. Data polymorphism and type inference mechanisms automatically take care of type resolution, type checking and type conversion, and make users unaware of their existence most of the time. A rich expression language makes it easy to parameterize many aspects of a model statically or dynamically. However, these mechanisms add much indirection overhead and therefore cannot be used directly in an implementation.

The code generation framework takes a model shown to meet certain design specifications through simulation and/or verification. Through model analysis—the counterpart of binding-time analysis in traditional use of partial evaluation for general purpose software, it can discover the execution context for the model and the components (called actors in Ptolemy terminology) contained within. It then generates the target code specific to the execution context while preserving the semantics of the original model. See Fig. 1, which follows notions used in [5].

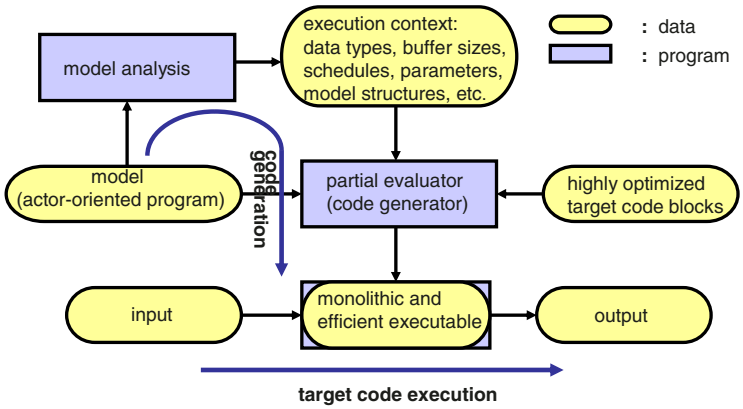

Fig. 1. Code generation with partial evaluation for actor-oriented programs

In this paper, C is our primary target language. In the generated target code, the variables representing the buffers in the input ports of each actor are defined with the data types discovered through type resolution. At the same time, if the model has a static schedule, then buffer sizes can be predetermined and defined

too (as arrays), thus eliminating the overhead of dynamic memory allocation. Through model analysis, the framework can also classify parameters into either static or dynamic. Static parameters have their values configured by users and stay constant during execution. Therefore there is no need to allocate memory for them and every time a static parameter gets used in the generated code, the associated constant gets substituted in. On the other hand, dynamic parameters change their values during execution. Therefore a corresponding variable is defined for each of them in the generated code. Most of models have static structures. The code generation framework takes advantage of this and eliminates the interfaces between components. In the generated code, instead of using a dozen or so indirection function calls to transfer data between components, a simple assignment is used, resulting in very efficient execution. For the MoCs that have static schedules, instead of dispatching actors based on the schedule, the schedule is hard-coded into the generated code, i.e., the code flow directly reflects the execution sequence, thus making it run much faster. Finally, for each actor that supports code generation, there is a corresponding helper which reads in pre-existing code blocks written in the target language. These target code blocks are functionally equivalent to the actor written in Java, the language used for Ptolemy II. The helper mechanism is elaborated in the next section.

2 A Helper-Based Architecture

A helper is responsible for generating target code for a Ptolemy II actor. Each Ptolemy II actor for which code will be generated in a specific language has one associated helper. An actor may have multiple helpers to support multiple target languages (C, VHDL, etc.), although we are concentrating on C in this paper.

To achieve readability and maintainability in the implementation of helpers, the target code blocks (for example, the initialize block, fire block, and wrapup block) of each helper are placed in a separate file under the same directory. So a helper essentially consists of two files: a java class file and a code template file. This not only decouples the writing of Java code and target code, but also allows using a target language specific editor while working on the target code, such as the C/C++ Development Toolkit in Eclipse.

For each helper, the target code blocks contained in the code template file are hand-coded, verified for correctness (i.e., semantically equivalent to the behavior of the corresponding actor written in Java) and optimized for efficiency. They are stored in the library and can be reused to generate code for different models. Hand-coded templates also retain readability in the generated code. The code generation kernel uses the helper java class to harvest code blocks. The helper java class may determine which code blocks to harvest based on actor instance-specific information (e.g., port types, parameter values). The code template file contains macros that are processed by the kernel. These macros allow the kernel to generate customized code based on actor instance-specific information.

2.1 What Is in a C Code Template File?

A C code template file has a .c file extension but is not C-compilable due to its unique structure. We use a CodeStream class to parse and use these files. Below are the C code template files for the Pulse and CountTrues actors (see Fig. 2).

```
// Pulse.c
/***preinitBlock***/
int $actorSymbol(iterationCount) = 0;
int $actorSymbol(indexColCount) = 0;
unsigned char $actorSymbol(match) = 0;
/**/

/***fireBlock***/
if ($actorSymbol(indexColCount) < $size(indexes)
        && $actorSymbol(iterationCount) == $ref(indexes, $actorSymbol(indexColCount))) {
    $ref(output) = $ref(values, $actorSymbol(indexColCount));
    $actorSymbol(match) = 1;
} else {
    $ref(output) = 0;
}
if ($actorSymbol(iterationCount) <= $ref(indexes, $size(indexes) - 1)) {
    $actorSymbol(iterationCount) ++;
}
if ($actorSymbol(match)) {
    $actorSymbol(indexColCount) ++;
    $actorSymbol(match) = 0;
}
if ($actorSymbol(indexColCount) >= $size(indexes) && $val(repeat)) {
    $actorSymbol(iterationCount) = 0;
    $actorSymbol(indexColCount) = 0;
}
/**/

// CountTrues.c
/*** preinitBlock ***/
int $actorSymbol(trueCount);
int $actorSymbol(i);
/**/

/*** fireBlock ***/
$actorSymbol(trueCount) = 0;
for($actorSymbol(i) = 0; $actorSymbol(i) < $val(blockSize); $actorSymbol(i)++) {
    if ($ref(input, $actorSymbol(i))) {
        $actorSymbol(trueCount)++;
    }
}
$ref(output) = $actorSymbol(trueCount);
/**/
```

A C code template file consists of C code blocks. Each code block has a header and a footer. The header and footer tags serve as code block separators. The footer is simply the tag "/**/".The header starts with the tag "/***" and ends with the tag "***/". Between the header tags are the code block name and optionally an argument list. The argument list is enclosed by a pair of parentheses "()" and multiple arguments in the list are separated by commas ",". A code block may have arbitrary number of arguments. Each argument is prefixed by the dollar sign "$" (e.g., $value, $width), which allows easy searching of the argument in the body of code blocks, followed by straight text substitution with the string value of the argument. Formally, the signature of a code block is

defined as the pair (N, p) where N is the code block name and p is the number of arguments. A code block (N, p) may be overloaded by another code block (N, p') where $p \neq p'$.[1] Furthermore, different helpers in a class hierarchy may contain code blocks with the same (N, p). So a unique reference to a code block signature is the tuple (H, N, p) where H is the name of the helper.

A code block can also be overridden. A code block (H, N, p) is overridden by a code block $(\tilde{H}, N, p)$ given that $\tilde{H}$ is a child class of H. This gives rise to code block inheritance. Ptolemy II actors have a well-structured class hierarchy. The code generation helpers mirror the same class hierarchy. Since code blocks represent behaviors of actors in the target language, the code blocks are inherited for helpers just as action methods are inherited for actors. Given a request for a code block, a CodeStream instance searches through all code template files of the helper and its ancestors, starting from the bottom of the class hierarchy. This mirrors the behavior of invoking an inherited method for an actor.

2.2 What is in a Helper Java Class File?

Helper classes are inherited from CodeGeneratorHelper. The CodeGeneratorHelper class implements the default behavior for a set of methods that return code strings for specific parts of the target program (init(), fire(), wrapup(), etc.), using the default code block names (initBlock, fireBlock, wrapupBlock, etc.). Each specific helper class can either inherit the behavior from its parent class or override any method to read code blocks with non-default names, read code blocks with arguments, or do any special processing it deems necessary.

2.3 The Macro Language

The macro language allows helpers to be written once, and then used in a different context where the macros are expanded and resolved. All macros used in code blocks are prefixed with the dollar sign "$" (as in "$ref(input)", "$val(width)", etc.). The arguments to the macros are enclosed in parentheses. Macros can be nested and recursively processed by the code generation helper. The use of the dollar sign as prefix assumes that it is not a valid identifier in the target language. The macro prefix can be configured for different target languages. Different macro names specify different rules for text substitutions. Since the same set of code blocks may be shared by multiple instances of one helper class, the macros mainly serve the purpose of producing unique variable names for different instances and generating instance-specific port and parameter information. The following is a list of macros used in C code generation.

$ref(name). Returns a unique reference to a parameter or a port in the global scope. For a multiport which contains multiple channels, use $ref(name#i) where i is the channel number. During macro expansion, the name is replaced by the full name resulting from the model hierarchy.

[1] All arguments in a code block are implicitly strings. So unlike the usual overloaded functions with the same name but different types of arguments, overloaded code blocks need to have different number of arguments.

$ref(name, offset). Returns a unique reference to an element in an array parameter or a port with the indicated offset in the global scope. The offset must not be negative. $ref(name, 0) is equivalent to $ref(name). Similarly, for multiport, use $ref(name#i, offset).

$val(parameter-name). Returns the value of the parameter associated with an actor in the simulation model. The advantage of using $val() macro instead of $ref() macro is that no additional memory needs to be allocated. $val() macro is usually used when the parameter is constant during the execution.

$actorSymbol(name). Returns a unique reference to a user-defined variable in the global scope. This macro is used to define additional variables, for example, to hold internal states of actors between firings. The helper writer is responsible for declaring these variables.

$size(name). If the given name represents an ArrayType parameter, it returns the size of the array. If the given name represents a port of an actor, it returns the width of that port.

2.4 The CountTrues Example

Fig. 2 shows a very simple model named CountTrues (notice it has the same name as one actor used in the model) in the synchronous dataflow (SDF) domain (In Ptolemy II, a domain realizes an MoC). In the model the Pulse actor produces "true" or "false" token and the CountTrues actor counts the "true" tokens. The CountTrues actor has its "blockSize" parameter set to 2, which means in each firing it reads 2 tokens from its input port and sends out a token recording the number of "true" tokens. When the model is simulated in the Ptolemy II framework, the produced result is shown on the right hand side of the figure (the model is fired 4 times because the SDFDirector's "iterations" parameter is set to 4). Below is the main function of the generated stand-alone C program.

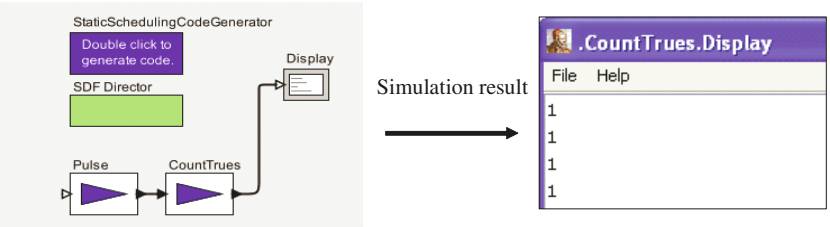

Fig. 2. The CountTrues model and its simulation result

```
......
static int iteration = 0;
main(int argc, char *argv[]) {
    init();
    /* Static schedule: */
    for (iteration = 0; iteration < 4; iteration ++) {
```

```
        /* fire Composite Actor CountTrues */
        /* fire Pulse */
        if (_CountTrues_Pulse_indexColCount < 2
                && _CountTrues_Pulse_iterationCount == Array_get(_CountTrues_Pulse_indexes_ ,
                _CountTrues_Pulse_indexColCount).payload.Int) {
            _CountTrues_CountTrues_input[0] = Array_get(_CountTrues_Pulse_values_ ,
                _CountTrues_Pulse_indexColCount).payload.Boolean;
            _CountTrues_Pulse_match = 1;
        } else {
            _CountTrues_CountTrues_input[0] = 0;
        }
        if (_CountTrues_Pulse_iterationCount
                <= Array_get(_CountTrues_Pulse_indexes_ ,2 - 1).payload.Int) {
            _CountTrues_Pulse_iterationCount ++;
        }
        if (_CountTrues_Pulse_match) {
            _CountTrues_Pulse_indexColCount ++;
            _CountTrues_Pulse_match = 0;
        }
        if (_CountTrues_Pulse_indexColCount >= 2 && true) {
            _CountTrues_Pulse_iterationCount = 0;
            _CountTrues_Pulse_indexColCount = 0;
        }
        /* fire Pulse */
        // The code for the second firing of the Pulse actor is omitted here.
        .....
        /* fire CountTrues */
        _CountTrues_CountTrues_trueCount = 0;
        for(_CountTrues_CountTrues_i = 0; _CountTrues_CountTrues_i < 2;
                _CountTrues_CountTrues_i++){
            if (_CountTrues_CountTrues_input[(0 + _CountTrues_CountTrues_i)%2]) {
                _CountTrues_CountTrues_trueCount++;
            }
        }
        _CountTrues_Display_input[0] = _CountTrues_CountTrues_trueCount;
        /* fire Display */
        printf("Display: %d\n", _CountTrues_Display_input[0]);
    }
    wrapup();
    exit(0);
}
```

In the code the $ref() and $actorSymbol() macros are replaced with unique variable references. The $val() macro in the CountTrues actor's code block is replaced by the parameter value of the CountTrue instance in the model. When the generated C program is compiled and executed, the same result is produced as from the Ptolemy II simulation.

3 Software Infrastructure

Our code generation framework has the flavor of code generation domains in Ptolemy Classic [13]. However, in Ptolemy Classic, code generation domains and simulation domains are separate and so are the actors (called stars in Ptolemy Classic terminology) used in these domains. In Ptolemy Classic, the actors in the simulation domains participate in simulation whereas the corresponding actors in the code generation domains participate in code generation. Separate domains (simulation vs. code generation) make it inconvenient to integrate the model design phase with the code generation phase and streamline the whole process.

Separate actor libraries make it difficult to maintain a consistent interface for a simulation actor and the corresponding code generation actor.

In Ptolemy II, there are no separate code generations domains. Once a model has been designed, simulated and verified to satisfy the given specification in the simulation domain, code can be directly generated from the model. Each helper does not have its own interface. Instead, it interrogates the associated actor to find its interface (ports and parameters) during the code generation. Thus the interface consistency is maintained naturally. The generated code, when executed, should present the same behavior as the original model. Compared with the Ptolemy Classic approach, this new approach allows the seamless integration between the model design phase and the code generation phase.

In addition, our code generation framework takes advantage of new technologies developed in Ptolemy II such as the polymorphic type system, richer variety of MoCs including hierarchical concurrent finite-state machines [14] which are well suited for embedded system design and discussed in Sect. 4.

To gain an insight into the code generation software infrastructure, it is worthwhile to take a look at how actors are implemented for simulation purposes. In Ptolemy II, the Executable interface defines how an actor can be invoked. The preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution. The initialize() method is assumed to be invoked once after the type resolution. The prefire(), fire(), and postfire() methods will usually be invoked many times, with each sequence of method invocations defined as one iteration. The wrapup() method will be invoked exactly once per execution at the end of the execution.

The Executable interface is implemented by two types of actors: AtomicActor, which is a single entity, and CompositeActor, which is an aggregation of actors. The Executable interface is also implemented by the Director class. A Director class implements an MoC and governs the execution of actors contained by an (opaque) CompositeActor.

The classes to support code generation are located in the subpackages under ptolemy.codegen (In Ptolemy II architecture, all the package paths start with "ptolemy"). The helper class hierarchy and package structure mimic those of regular Ptolemy II actors. The counterpart of the Executable interface is the ActorCodeGenerator interface. This interface defines the methods for generating target code in different stages corresponding to what happens in the simulation. These methods include generatePreinitializeCode(), generateInitializeCode(), generateFireCode(), generateWrapupCode(), etc.

CodeGeneratorHelper, the counterpart of AtomicActor, is the base class implementing the ActorCodeGenerator interface. It provides common functions for all actor helpers. Actors and their helpers have the same names so that the Java reflection mechanism can be used to load the helper for the corresponding actor during code generation. For example, there is a Ramp actor in the package ptolemy.actor.lib. Correspondingly, there is a Ramp helper in the package ptolemy.codegen.c.actor.lib. Here c represents the fact that all the helpers under ptolemy.codegen.c generate C code. Assume we would like to generate

code for another target language X, the helpers could be implemented under ptolemy.codegen.x. This results in an extensible code generation framework. Developers can not only contribute their own actors and helpers, but also extend the framework to generate code for a new target language.

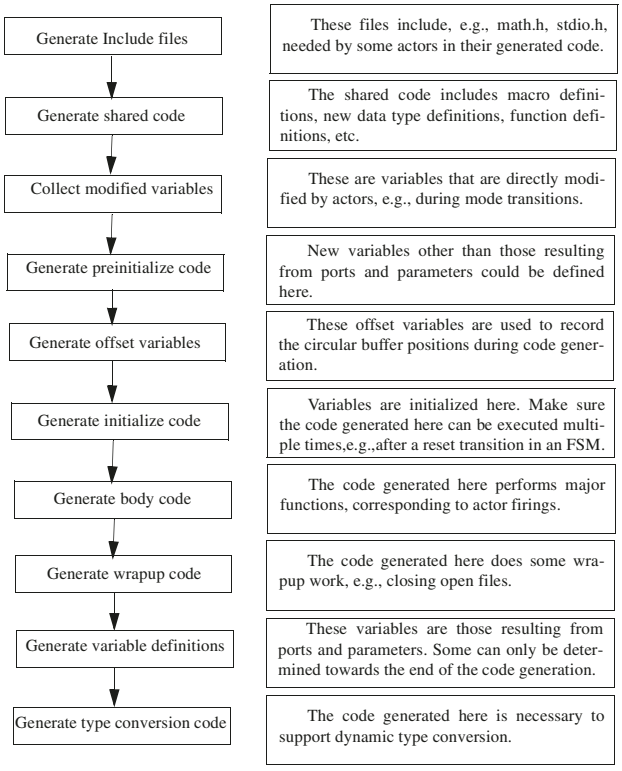

Fig. 3. The flow chart of the code generation process

To generate code for hierarchically composed models, helpers for composite actors are also created. For example, the most commonly used composite actor is TypedCompositeActor in the package ptolemy.actor. A helper with the same name is created in the package ptolemy.codegen.c.actor. The main function of this helper is to generate code for the data transfer through the composite actor's interface and delegate the code generation for the composite actor to the helper for the local director or the helpers for the actors contained by the composite actor. Since a director implements an MoC (called a domain in Ptolemy terminology), a helper is created for each director that supports code generation. These director helpers generate target code that preserves the semantics of MoCs. Currently, the synchronous dataflow domain (SDF), finite state machines (FSM), and heterochronous dataflow domain (HDF) support code generation (see Sect. 4 for more details).

Finally the StaticSchedulingCodeGenerator class is used to orchestrate the whole code generation process. An instance of this class is contained by the top level composite actor (represented by the blue rectangle in Fig. 2). The code generation starts at the top level and the code for the whole model is generated hierarchically, much similar to how a model is simulated in Ptolemy II.

The flow chart in Fig. 3 shows the whole code generation process step by step. The details of some steps are MoC-specific. Notice that the steps outlined in the figure do not necessarily follow the order the generated codes are assembled together. For example, only those parameters that change values during the execution need to be defined as variables. Therefore those definitions are generated last after all the code blocks have been processed, but placed at the beginning of the generated code. Our helper based code generation framework actually serves as a coordination language for the target code. It not only leverages the huge legacy code repository, but also takes advantage of many years and many researchers' work on compiler optimization techniques for the target language, such as C. It is accessible to a huge base of programmers. Often new language fails to catch on not because it is technically inferior, but because it is very difficult to penetrate the barrier established by the languages already in widespread use. With the use of the helper class combined with target code template written in a language programmers are familiar with, there is much less of a learning curve to use our design and code generation environment.

4 Domains

SDF: The synchronous dataflow (SDF) domain [15] is a mature domain in Ptolemy II. Under SDF, the execution order of actors is statically determined prior to execution. This opens the door for generating some very efficient code. In fact, the SDF software synthesis has been studied extensively. Many optimization techniques have been designed according to different criteria such as minimization of program size, buffer size, or actor activation rate. We built the support for SDF code generation to test our framework and use it as a starting point to explore code generation for other domains.

FSM: Finite state machines (FSMs) have a long history. We use hierarchical concurrent finite state machines [14]. In Ptolemy II, an FSM actor can do traditional FSM modeling or specify modal models. In traditional FSM modeling, an FSM actor reacts to the inputs by making state transitions and sending data to the output ports like an ordinary Ptolemy actor. The FSM domain also supports the *charts formalism with modal models. In Fig. 4, M is a modal model with two modes. Modes are represented by states (rendered as circles in the figure) of an FSM actor that controls mode switching. Each mode has one or more refinements that specify the behavior of the mode. A modal model is constructed in a ModalModel actor having the FSMDirector as the local director. The ModalModel actor contains a ModalController (inherited from FSMActor) and a set of Refinement actors that model the refinements associated with

states and possibly a set of transition refinements. The FSMDirector mediates the interaction with the outside domain, and coordinates the execution of the refinements with the ModalController. We created helpers for FSMDirector, FSMActor, ModalController, ModalModel, Refinement and TransitionRefinement and are capable of generating C code for both traditional FSM modeling and modeling with modal models.

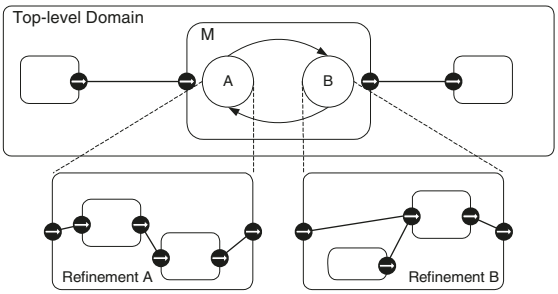

Fig. 4. A modal model example

HDF: In Fig. 4, if the top level domain and the domains inside the refinements are all SDF, then we get the very interesting heterochronous dataflow (HDF) domain. An HDF model allows changes in port rates (called rate signatures) between iterations of the whole model. Within each iteration, rate signatures are fixed and an HDF model behaves like an SDF model. This guarantees that a schedule can be completely executed. Between iterations, any modal model can make a state transition and therefore derives its rate signature from the refinement associated with the new state. The HDF domain recomputes the schedule when necessary. Since it is expensive to compute the schedule during the run time, all possible schedules are precomputed during code generation.

The HDF domain can be used to model a variety of applications that SDF cannot easily model. For example, in control application, the controlled plant can be in a number of operation states, requiring a number of control modes. In communication and signal processing, adaptive algorithms are used to achieve optimal performance with varying channel conditions. In all these applications, the HDF domain can be used to model their modal behaviors, leading to implementations that can adjust operation modes according to the received inputs, while still yielding static analyzability due to finite number of schedules.

5 Conclusion

This paper describes a code generation framework for actor-oriented models using partial evaluation. It uses a helper-based mechanism to achieve modularity, maintainability, portability and efficiency in code generation. It demonstrates design using high level abstraction can be achieved without sacrificing performance. The code generation framework is part of Ptolemy II 6.0.1 release. It can

be downloaded from the Ptolemy project website at EECS, UC Berkeley. The software release includes various demos to highlight the features of the code generation framework. We are currently exploring code generation for other MoCs suited to embedded system design. We are also testing the capabilities of the code generation framework with more complicated applications.

Acknowledgements. This paper describes work that is part of the Ptolemy project, which is supported by the National Science Foundation (NSF award number CCR-00225610), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments and Toyota.

References

1. E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33-42, May 2006.
2. http://www.microsoft.com/whdc/driver/wdf/WDF_facts.mspx
3. W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, 2002 Springer-Verlag LNCS, Grenoble, France, April, 2002.
4. http://www.mathworks.com/products/simulink/
5. N. D. Jones, C. K. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall, June 1993.
6. E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 251-263, October 2002.
7. C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, vol.1, no.1, Jan. 1991, pp. 21-69.
8. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, vol.11, no.3-4, Oct.-Nov. 1991, pp. 217-42.
9. L. O. Andersen. Partial evaluation of C and automatic compiler generation. In *4th International Conference CC'92 Proceedings*. Springer-Verlag. 1992, pp. 251-7.
10. U. Schultz. Partial evaluation for class-based object-oriented languages. In *Proceedings of Symposium on Programs as Data Objects (PADO)*, number 2053 in Lecture Notes in Computer Science. Springer-Verlag, May 2001.
11. R. V. Ommerling. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):7885, March 2000.
12. J. Tsay. A Code Generation Framework for Ptolemy II. ERL Technical Memorandum UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000.
13. 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.
14. A. Girault, B. Lee, and E. A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, Vol. 18, No. 6, June 1999.
15. E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, September, 1987.