An Extensible and Retargetable Code Generation Framework for Actor Models

Man-Kit Leung

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-187
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-187.html

December 19, 2009

An Extensible and Retargetable Code Generation Framework for Actor Models

by

Man-Kit Leung

Submitted in partial satisfaction of the requirements for
the degree of Master of Science (Plan II)

in

Electrical Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2009

The report of Man-Kit Leung is approved:

Professor Edward A. Lee (Research Advisor) Date

Professor Koushik Sen (Second Reader) Date

University of California, Berkeley

Fall 2009

An Extensible and Retargetable Code Generation Framework for Actor Models

Man-Kit Leung

Abstract

Model-based code generation (MBCG) is a new field of research that arises from the advancement of high-level modeling languages. It is the syntactical transformation of a source model into some textual representation or program. Its applicability ranges from accelerating execution, targeting various platforms to enhancing model interoperability. MBCG is a key agent in sharing models across different research communities and maximizing the impact of the model-based design methodology. However, because of the raised level of abstractions, code generation in a model-based environment brings up a new set of problems beyond traditional compiler research.

Using Ptolemy II as an exemplar modeling environment, this report explores the challenges and problems that exist for model-based code generation. The report describes a reusable code generation framework built using two simple design patterns, code template and adapter, that are well-suited for an actor-based environment. The extensible infrastructure helps simplify the process for future experiments, especially for new models of computation (MoCs). The report tackles the challenges posed by configurable higher-order actors. It also demonstrates a system for incorporating multiple datatypes and specializing polymorphic functions to increase code efficiency.

Acknowledgements

Before diving into the technical details of the report, I would like to express my gratitude to several people who have made my graduate experience possible. First, I want to thank Professor Edward A. Lee, my graduate advisor, who has been a role model for me in this graduate program. His vision, guidance and sheltering have accompanied me throughout my research. I would also like to thank Professor Koushik Sen for serving as my second reader of this report and giving his invaluable insights. I want to thank Dr. Hiren Patel for sharing office space with me. His encouragements and companionship have helped me weather through hard times. He has also helped revise my writing in this report. Next, I would like to thank the group of people who I worked with in the Ptolemy Pteam. Christopher Brooks, Director of our CHESS Research Center, has provided me wonderful technical support and helped tremendously in releasing the technology described in this report. I also want to thank the past, present and future Codegen Pteam, which includes Gang Zhou, Teale Fristoe, Jia Zou, Isaac Liu, Bert Rodiers, Ben Lickly and Shanna Forbes. All of whom have shared with me the burdens of solving design and implementation problems.

Next, I would like to thank a group of people who have been behind-the-scenes for my accomplishments. I especially dedicate my personal thanks to Pastor Helen and my dear friends, Jeffrey and Irene, for their constant prayer and support. I have to thank my sister,

Caroline, for inspiring my interests in electrical engineering. I deeply thank my parents, Kwok Leung and Oiwan Cheung, for bringing me into this world and the United States. They have sacrificed their own freedom in many ways so that I can have enough to achieve my dreams. Lastly, I thank God my Lord for giving me this extraordinary experience in life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As model-based design is becoming the commonly accepted methodology in today's system design, the emphasis is placed in building useful tools and artefacts. Building models helps a designer to better understand the dynamics of the real system. A high-level model abstracts away the incidental details, often pertaining to implementation, while keeping the essence of the system. It allows architects to reason and make conjectures about system behaviors.

However, adopting this methodology presents major challenges. On one hand, researchers are faced with the problem of how to maximize the utility of a model. Enhancing models with *executability* is one mainstream approach. It equips models with dynamic semantics so that a designer can conduct simulation and visualize preliminary results. For instance, many recent efforts in the Unified Modeling Language (UML) community have been devoted to extending UML 2.0's Activity Diagrams with executable semantics [1]. A second challenge is building the *support of tools* that handle various models. The grand vision of modeling is to give users the freedom to choose the right abstractions that represent a simplified view of the system. Ideally, engineering teams can

build separate models to represent different views of the same system. Models can be transformed and serve as a unit of communication to pass information between engineering teams. Models can even be combined to give richer details about a system. The models are eventually synthesized as a part or whole of the real system using automatic code generation. Tools are necessary for each of these processes in order to maximize productivity and minimize errors.

The effort in building any such modeling tool is non-trivial. Each tool specializes in a particular domain of application, such as modeling, simulation, testing or verification. Often, a tool implements its own **domain-specific language (DSL)**, a language that is suitable for its domain of use. It is hard to standardize the different DSLs and come up with an one-size-fit-all modeling language. To alleviate this problem, it becomes increasingly important to build tools that address *interchangeability*, so that models can be exchanged and reused by multiple tools. It enables ideas such as co-modeling, co-simulation, and co-verification where tools interact and work cooperatively with one another.

Code generation, in particular, plays a key role in model-driven engineering. It is the process that transforms a model graph to textual representation. In today's electronic system design, automatic transformation can be found in various processes and applications. Logic synthesis in electronic design automation (EDA) contains many such examples. Tools, such as Espresso [2], are used to automate the optimization of circuit logic. Logic circuits then pass through the process of *technology mapping* which combinatorially synthesizes a circuit with fabricated elements [3]. Such combinatorial approach is also found in the code generation of programming languages. The GNU Compiler Collection (GCC), for example, generates code by searching for an optimal tree covering, given the abstract syntax tree (AST) of an input program. Heuristics, such as coagulation, are often

used because searching for the optimal solution is NP-hard.

Java and Microsoft's .NET framework innovate a technique called **Just-In-Time (JIT)** compilation. It is an innovative use of code generation. JIT introduces the process of code generation into runtime by dynamically converting bytecode into native code. By combining with runtime program analysis, it helps improve the execution performance. Moving to areas of mechatronics and safety-critical applications, we can see the emergence of model-level synthesis. High-level, often graphical, languages such as SCADE, Simulink, and LabVIEW provide the ability to synthesize C code from design models. The vision is to increase the amount of confidence and trust to put in these high-level synthesis tools. We have seen partial success in this direction. SCADE's code generator, in particular, has been certified for the Federal Aviation Administration (FAA) as a heavily used development tool in aerospace and defence applications [4].

Model-based code generation is often closely tied with **template metaprogramming**, where a programming language is used to manipulate code written in another programming language. The manipulator language is often referred to as the **metalanguage**, while the target or **object language** refers to the one being operated on. In a way, the object language is treated as first-class data in a metaprogramming program using a so-called **template**. A template is interpreted at compile time to produce programs that are further executed by other platforms. The two-phase execution helps incorporate information from multiple sources at compile time and give programmers the flexibility to handle various conditions. There exist several popular template-driven languages frameworks, such as openArchitectureWare (oAW), Java Emitter Template (JET), and Velocity. oAW is a tool suite that specializes in model-driven design. Its core feature, Xpend, is a user-friendly template language that enhances integration of tools. JET is a template language

tool shipped with the Eclipse Modeling Framework (EMF). Velocity is a template language used by the Apache Group to separate presentation logic from the business tiers in web applications.

As we shall see, a template language is only one necessary part for model-based code generation. A *description of the target language and platform* is also needed in order to help the code generator to make decisions relating to code and performance optimizations. In addition, a code generator requires a *description of the source modeling language* so that it can adapt to the modeling language and effectively utilize model data. **Metamodeling** is one approach to solve this adaptation problem. It unifies the syntax in both the code generator and the modeling language. It explicitly lays out the relationship and allowable operations for each type of model objects. Finally, *static analysis* and *partial evaluation* are performed after combining the modeling and architectural information.

1.1 Actor-Oriented Modeling

Actor-oriented modeling is a useful tool for modeling and reasoning about concurrent execution. An actor model consists of basic building blocks called **actors**. Each actor has a set of ports and parameters. A **port** is the primary channel for an actor to communicate with the outside world. Ports are connected together to enable data exchange between actors. A port can be further classified as either an input or an output. An input port denotes the receiving end of a connection, while an output port denotes the sending end. **Parameters** are explicit information tagged with an actor. They are often static information supplied by the model designer to specialize the behavior of an actor. Hierarchical composition of actors is also possible. It helps create an abstract view of a model by encapsulating refinements in sub-models. The flexible conceptual framework allows

for domain-specific abstractions and analyses. At the same time, its ability to be graphically represented has attracted many interests in extending its research and tools development. Tools such as Simulink, from MathWorks, LabVIEW, from National Instruments, SystemC, component and activity diagrams in SysML and UML 2 [5, 6, 7], and a number of research tools such as ModHel'X [8], TDL [9], HetSC [10], ForSyDe [11], Metropolis [12], and Ptolemy II [13] are based on actor modeling. In particular, the experiment in this report is built in the Ptolemy II environment.

Ptolemy II [14] is an extendible software platform used in the design, modeling and simulating of concurrent software. It is an open platform that allows designers to create and experiment with new actors. Its documentations and source are freely distributed on the public web for academic research and commercial extensions [15]. This choice in using Ptolemy II as our experimental platform is not arbitrary. Although actor-oriented modeling serves as a common syntax for models, it says little about the semantics of models. The specific details about actor scheduling and how data is exchanged are left open for interpretation. The interpretation of the model semantics is called a **model of computation (MoC)**. Unlike most other tools, Ptolemy II offers a flexible way to experiment with different MoCs. Each MoC is modularly implemented by a component called the **director**. To illustrate its core features and capabilities, let's look at a Ptolemy II model shown in Figure 1.1. It models the gravitational forces between 6 particles of equal mass. I enhanced the model, originally by authored by Professor Edward A. Lee, with a collision detection sub-system. For simplicity, I will use the original version for illustration.

It has a director called PN Director and four parameters, initialPositions, initialVelocities, numberOfBodies, and stepSize. The PN Director implements the Kahn Process Network semantics, which we will discussed in detail later. Each of the parameter is used for customizing the actors in

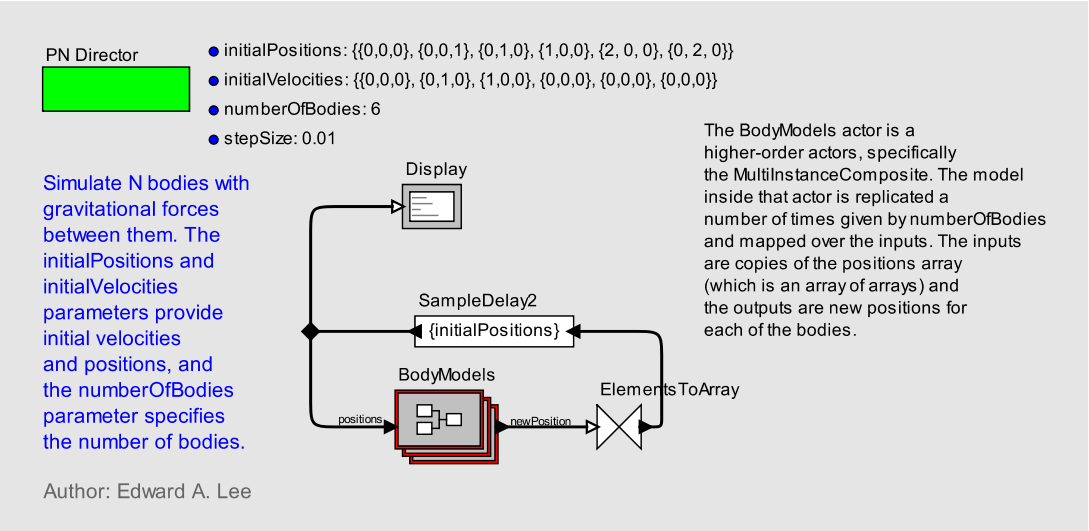

Figure 1.1: A model of particles and their gravitational forces.

the model. There are four actors, Display, SampleDelay, ElementsToArray, and BodyModels. Display, SampleDelay, and ElementsToArray are **atomic actors** whose behavior is purely specified by the actor designer and configurable only through parameters. ElementsToArray produces a stream of array tokens by combining multiple input streams of data tokens. SampleDelay transforms its input stream of tokens by inserting initial, or sometimes called prefix, token(s), and Display is a data sink that is used to visualize data tokens. On the other hand, BodyModels is a higher-order actor called MultiInstanceComposite that is *refined* by a sub-model consisting of other actors. A property of the model that is not immediately obvious is the difference of the **connection width**, or number of channels. The width between BodyModels and ElementsToArray is 6 because there are 6 instances of the BodyModels refinements, specified by the numberOfBodies parameter. ElementsToArray combines one token from each input channel into an array token. Thus, the connection width between ElementsToArray and SampleDelay is 1. **Single ports** that are solid black triangles only

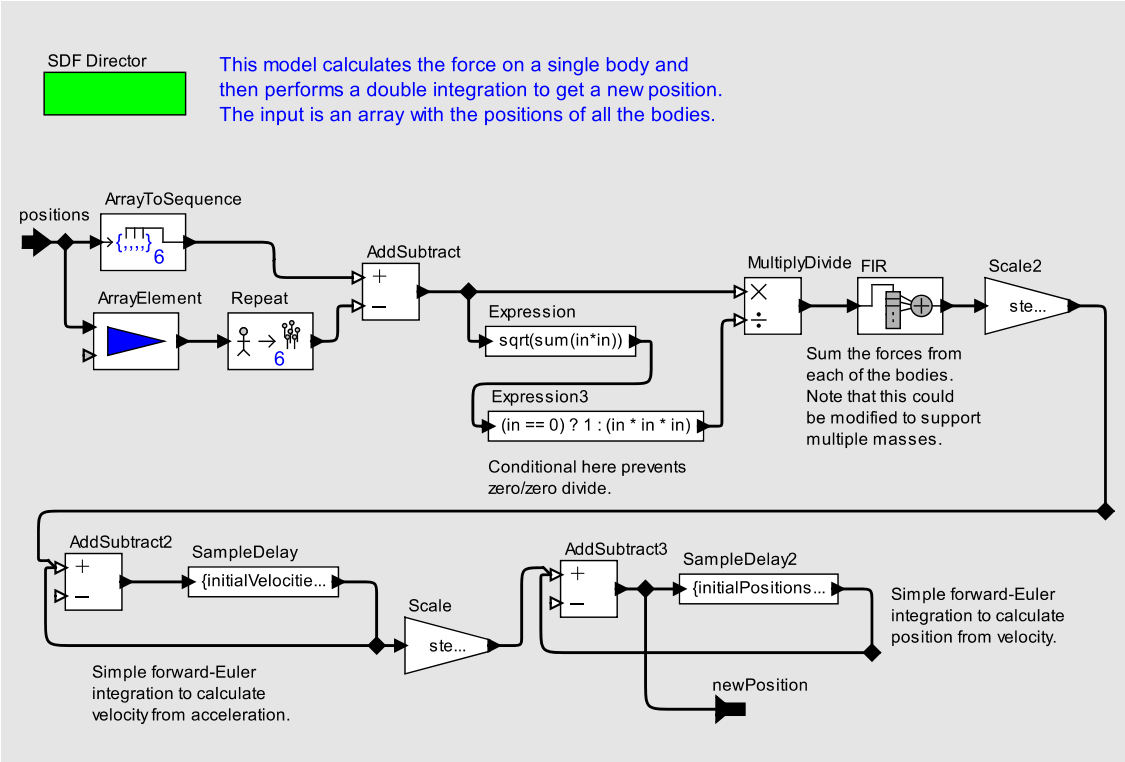

Figure 1.2: The refinement model of a particle's dynamics.

accept connections of width 1, and **multiports**, white filled triangles with black outline, can accept connections of any width. In general, connection widths are inferred from user parameters and actors. Users are prompted to specify width only when there is ambiguity.

Figure 1.2 shows the refinement model of `BodyModels`. The model computes the combined gravitational forces acting upon a particle and two explicit integrations to simulate the particle's position. All 14 actors (white rectangular and triangular containers) are atomic actors. `ArrayToSequence`, `Repeat`, and `SampleDelay` are flow control actors that manipulate the token stream without changing the token values. The rest of the actors perform a specific function on the incoming token values. In particular, there are two instances of `Expression`, which is a

highly configurable actor. Users can use the Ptolemy expression language [16] to specify computation for the input values. Section 3.1.4 will discuss how to generate code for the `Expression` actor.

A subtle property in the model is **datatype**. For example, the datatype of ArrayToSequence's input port is known to be array type, and its output port is of the type of the input array elements. Same is true for ArrayElement. The type of Repeat's input port is the same as its output port. Scale, AddSubtract, and MultiplyDivide are *polymorphic* actors that can operate over multiple datatypes. Ptolemy II has a sound type system [17] that statically infers the datatype of model components and gives precise errors when there exists conflicts. The model has a `SDF Director` which implements the Synchronous Dataflow (SDF) semantics [18]. It provides efficient execution for synchronous data computation. Next, we will discuss SDF in more detail along with a survey of other MoCs.

1.2 Models of Computation

A *model of computation* (MoC) defines the semantics for a model. It is a set of rules that govern the interaction between actors. The Ptolemy Project [19] is dedicated to defining and analyzing MoCs. Several models of computation have been proposed and studied. In particular, Lee and Sangiovantelli proposed the *tagged signal model* that provides a common framework to analyze MoCs [20]. The key observation is that most MoCs differ in their notion of *ordering* and *time*. Let's describe in the following several of the representative MoCs.

Dataflow is a one of the most studied and common model of computation, in which actors communicate purely by exchanging data tokens through ports. Execution of dataflow models is

purely data-driven and often in the absence of time. Several specialized forms of dataflow have been proposed. **Synchronous Dataflow (SDF)**, in particular, allows static scheduling of actor execution and allocation of buffer sizes. SDF has a notion of a minimal schedule which it iterates. It requires each input and output port to declare its consumption and production rate, respectively. It precomputes the schedule using the specified rates. Its scheduling decidability is particularly suitable for execution on parallel hardware. Pino et al. discovered a precedence graph formalism that systematically exposes the parallelism in SDF models for targeting DSP processors [21]. Modal behavior can be combined with SDF through composition with finite state machine (FSM). This allows model to express variable rate communication via state transition. This model of computation, consisting of hierarchical composition of SDF and FSM, is called the **Heterochronous Dataflow (HDF)** [22]. Each state in a HDF model can be refined by a SDF sub-model. There can be multiple execution schedules, one for each combination of state refinements that are activated in the FSMs. Since the number of states is finite and constant, it is possible to precompute all possible firing schedules. Girault and Lee has shown that HDF schedulability is decidable [22]. Code generator can encode the schedules statically in the generated program and minimize run-time overhead. Buck proposed an alternative dataflow formalism called the **Boolean Dataflow (BDF)** [23]. It introduces the BooleanSwitch and BooleanSelect actors whose production and consumption rates are driven by the data value. He has shown that this extension alone makes BDF Turing complete. **Dynamic Dataflow (DDF)** [24] is yet a more general form of dataflow formalism. Each actor in a DDF model has a set of firing rules. An actor is allowed to fire whenever one or more of these firing rules are enabled.

Kahn Process Network (KPN) is a formalism that models asynchronous message passing. It is similar to dataflow in that actors communicate purely through data tokens. Data tokens

are passed via FIFO buffer queues and thus consumed in order. Actors can execute freely as long as input tokens are available. They can be viewed as functions that transform streams of input tokens to streams of output tokens. We will discuss KPN with further details in Section 3.2.2 which illustrates the process of generating programs that preserves KPN semantics.

Sychronous/Reactive (SR) is a formalism that models the behavior of a set of signal values at discrete clock ticks. Each signal takes on a fixed value at each clock tick. *Absent* is a special value that is valid under the SR semantics. The SR semantics is the basis for synchronous languages such as Esterel [25], Signal [26], and Lustre [27]. A SR program may fire the same actor multiple times, within a single clock tick, until all signals reaches a fixed point. It is up to the compiler to optimize the scheduling of actor execution. In most cases, it is assisted by static analysis. Causality analysis [28], in particular, helps to identify actors that are dependent on each other and thus allows a static order of actor firing. Edwards also proposes in his thesis a complete execution algorithm using clustering graphs that combines static and dynamic scheduling [29].

Giotto is a time-driven formalism that is behind many current industrial tools and protocols. It is a suitable model of computation for real-time applications. Actors share a global notion of time. Actor execution can be statically scheduled at predetermined time. Thus, communication is implicitly synchronized, and resources are conveniently shared without an explicit locking mechanism. However, keeping a notion of time can be difficult and expensive to implement. The problem often involves minimizing misalignments of clocks, time drift and jitter, which decrease the effective time for sharing. More importantly, they may cause real-time constraint violations and thus undermine reliability if these defects or errors are not bound. The Time-Division Multiple Access (TDMA) protocol, used in many 2G cellular systems, follows this time-driven semantics.

It discretizes a signal into repeating periods of time slots. Multiple users may share the same frequency channel through their assignment of time slots. The Time-Triggered Protocol (TTP), which is based on TDMA, is capable of tolerating and recovering from some time faults through hardware redundancy. Other industrial standards that draw heavily from the Giotto model of computation include FlexRay, Time-Triggered Architecture (TTA) and Time Definition Language (TDL).

Discrete Event (DE) is a event-driven formalism where actors communicate through events. Each event consists of a data value and a time stamp. All events generated in the system are totally ordered on a real-valued time line. Many notable hardware simulators and languages assume the DE semantics [17]. In particularly, VHDL and Verilog are examples of such, although they assume only integer-valued time. DE can be generalized to variable time increments, including zero-time increments such as the case in super-dense time.

1.3 Model-Based Code Generation

Model-based code generation (MBCG) is a new field of research that arises from the advancement of high-level modeling languages. Modeling languages are attracted to the idea of code generation which enables them to be used as implementation languages. Code generation effectively synthesizes implementation code from design models. Ideally, it can extend the use of the same model in multiple stages in the design process. It can also translate a model into different formats so that multiple tools can share the same model.

The generative programming community (GPCE) focuses on generalizing techniques and tooling that can be applied across multiple domain-specific languages. It borrows heavily from metaprogramming and composable programming abstractions such as aspects and features. Com-

piler tools such as AspectJ [30] and AHEAD [31] are the de facto standards for this programming paradigm. It involves the use of a template language, metamodels of the source modeling language, and some description of the target language and platform.

The transformation to code is exposed to the user through a template language that manipulates a target (implementation) language. The Model-to-Text (M2T) development efforts specifically focus on advancing the capability and usability of such template languages. It has resulted in several widely used template languages such as OpenArchitectureWare (oAW), Java Emitter Template (JET), and Velocity. The metamodel of the source modeling language is used to help weave the model syntax into the templates. The user can then integrate information from model objects using the template language. The system may also leverage a language that describes target platforms. Such can be provided through an architecture description language (ADL). There are several ADLs that already provide extensive tooling such as Acme and Architecture Analysis Design Language (AADL). My report is a special case study that describes a light-weight framework that generates retargetable code for the Ptolemy II modeling environment. At the same time, code generation for models can be extended to new areas of application. For example, a model-based code generator may be used as a model transformation tool, which generates model specifications for other tools. It enhances model interoperability by presenting the same model in a different syntax while preserving its semantics.

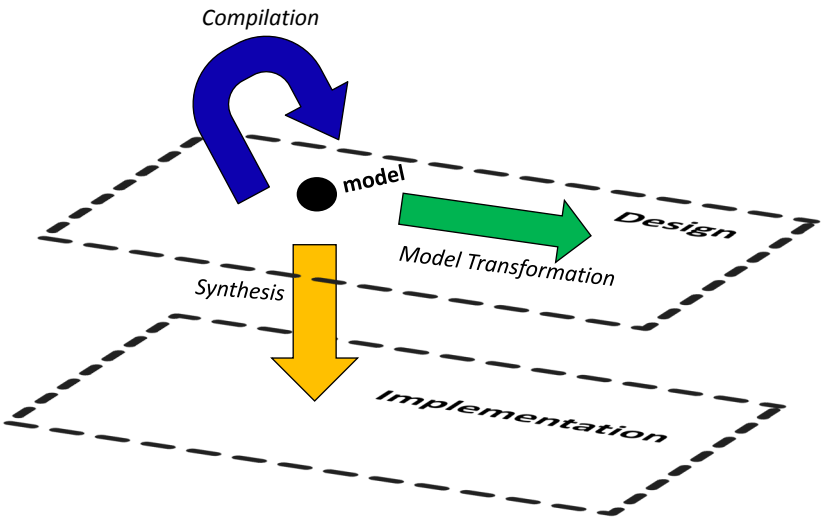

Figure 1.3: Application of code generation in the design flow.

Chapter 2

Problem Definition

This chapter describes the overall task of transforming model to code. It attempts to provide a mathematical basis. In particular, code generation is formulated as a basic graph problem of topologically sorting a set of code segments. Our framework provides two key abstractions, *adapter* and *code block*, to create code segments and specify the ordering relationship between them.

2.1 Overview

Let us look at an overview of the problem. High-level modeling languages inevitably introduces data structures and semantics that broaden the space between the design language and implementation. The immediate effect of this widened space is an increase of complexity. The translation or compilation process now becomes harder and thus requires more elaborate analyses. The complexity can be looked at through three measures: *implementability*, *completeness*, and *semantic correspondence*.

Implementability measures the different ways a *design model* can be implemented. It is a ratio between a model and its corresponding programs. For example, 1:25 indicates that a particular model can be represented by 25 different valid programs. Translation of binary to machine code is 1:1 implementable, since there is a unique machine code for every binary instruction. However, implementability is infinite in general. This quantification requires considerable restrictions in order to reduce the set of programs and measure the exact number of distinct programs. We will use implementability as an abstract concept to help us understand the complexity in MBCG. The problem of model-based code generation is not totally structured, and thus *a correct transformation in MBCG is not unique*. There is often more than one feasible programs that expresses the same design. The code generator needs to make many decisions in order to generate the final program, which is one instance out of many possible choices. It may leave some of these decisions up to the user through parametrizations. In short, implementing a model-based code generator requires considering many variables. The increased complexity between modeling language and implementation code often increases the implementability of a model but creates a larger solution search space for the code generator.

Completeness refers to the degree to which the *modeling language* is amenable to code generation. By raising the level of abstraction, the expressiveness of the language is increased. Constructs or semantics may not be expressible by the lower-level language. This is the case for hardware description languages, such as VHDL and Verilog. In these languages, there is the notion of *syntesizable* constructs, the subset of the language that can be compiled and generated to hardware. The rest of the language constructs are *simulated*, or non-synthesizable. This incompleteness comes from the mismatch between language abstractions and the support provided by the under-

lying platforms. Since an actor-based language is made of actor components, the completeness of code generation is quantifiable by counting the number of supported components. It is desirable to have a code generator that is incrementally extensible such that it becomes more complete over time. Given such a framework, development efforts can be accumulated and reused over time.

The third source of complexity comes from **semantic correspondence** between the design model and the generated code. Model-based code generation transforms a given model to code. It is a mapping, or conversion, from the *model space* to *code space*. Correspondence, or *semantics preservation*, is realized through a series of syntactic transformations. The degree of correspondence between a model and the generated program often depends on applications, or how the program is used. Generally, high degree of correspondence increases the complexity of the transformation because more variables need to be considered in the process.

Our framework specifically targets the completeness and correspondence problems. Implementability still poses many open problems and thus requires further study of model semantics. The output of our code generator is a single program file, as opposed to a library of files. This is for the ease of compiling at the command-line. It helps decrease the burdens on the user side. It is rather a pragmatic design decision. We also define for the rest of the writing the notion of **generate-time**, which distinctly refers to the operations performed by our code generator, as opposed to compile-time and runtime.

2.2 A Basic Formulation using Graph

The underlying intuition of our framework is a formulation using graph. Any given program can be seen as a series of code segments. Assume that we have a regular graph $G = (V, E)$

Appear-first	**Appear-later**
Header directive	Reference
Variable declaration	Reference
Function declaration	Reference
Statement 1	Statement 2
Definition	Instantiation
Keywords	Predicate expressions
Function invocation	Argument expressions

Table 2.1: Example of ordered constructs.

where each $v \in V$ is a segment of code, and each $e \in E$ is a precedence relation between two code segments. For example, if there is an edge going from *codeBlock1* to *codeBlock2*, then *codeBlock1* must precede, or be printed before, *codeBlock2* in the generated program. The edges specify the set of ordering constraints on the code blocks in the final program. Any feasible program to generate is therefore a topological sort of the graph, with respect to the given edges. The problem now becomes how to come up with these code blocks and specify their relationship. This is similar to the idea of basic blocks in a control flow graph (CFG), except that our edges correspond to ordering on appearance rather than execution.

This formulation is rooted in the experience of generating C programs [32]. For example, the familiar line `#include <stdio.h>` in C tells the preprocessor to bring in the content of `stdio.h`, and the directive should appear before any references to the content within `stdio.h`. To see this in the context of graph, the directive is treated as a single code block (node), say d. If there are other code blocks that, say for example, contain calls to `printf()`, then there will be an edge going from d to each of these code blocks. In fact, many programming constructs found in a textual program requires explicit ordering. Table 2.1 shows a brief list of these constructs.

2.3 Designing a Robust Framework

Our framework borrows from the *generic programming* paradigm in which templates are used as the fundamental abstraction [33]. Information is combined with the template at generate time to produce source code. Many notable languages support this template-style programming. For example, C++ provides class and function templates that can be used as patterns for creating instances of different types when the parametrized type information is supplied. Java also begins to support generics, which is a similar construct, starting from version 1.5. **Template specialization** is a common term that refers to the process of bounding template parameters with data. In our framework, specialization is an elaborate process that analyze and incorporates data from different sources.

The major design goal behind our framework is *robustness*. This means small extension to the modeling language requires little changes to the code generation framework. This requires a system that can be scaled up systematically. The key idea is to modularize our framework. At a high level view, there are four major packages of components that makes up our code generation framework, as shown in Figure 2.1.

Actors, directors, and datatypes are primitive classes of components in our modeling language, in this case, Ptolemy II. Targets are an orthogonal class of components that is transparent to the modeling environment. They exist for the purpose of retargeting, which we shall discuss in Chapter 4. Each of these packages consists of a set of code templates and adapters that follows a specific format within the package. When a new component is added in Ptolemy II, we can easily extend code generation for these components by defining the corresponding code template and adapter. The interface of the code template and adapter in each package provides intuitive guidance

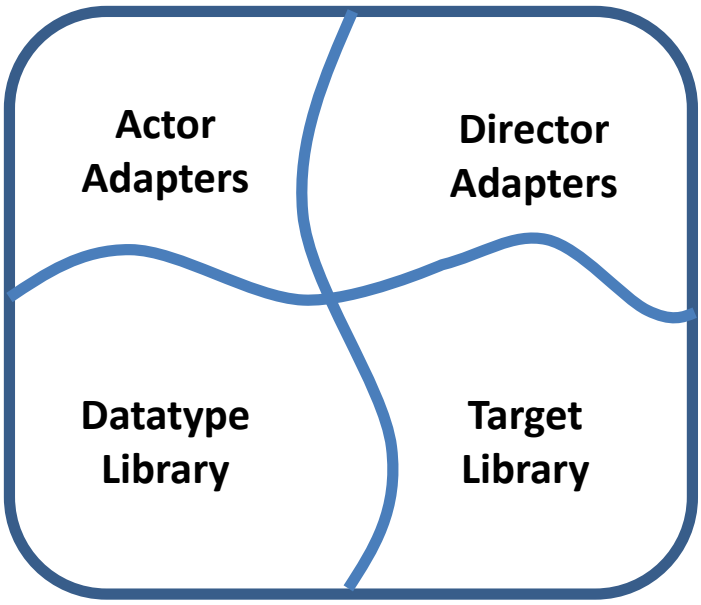

Figure 2.1: The four packages of the code generation framework.

that helps map component behaviors to implementation code. Thus, designers for new components can easily achieve full system code generation with this modular framework.

2.3.1 Code Templates and Code Blocks

Code templates are passive (non-executable) objects used by adapters to instantiate code blocks. There is a code template for every adapter. A code template contains a set of code blocks written in the target language. A code block provides an abstraction over the target language. It allows adapters to manipulate code by performing simple operations over the code blocks.

There are four operations provided by code blocks: instantiation, precedence, addition, and composition. **Instantiation** is a unary operation that creates a new instance of a given code block. It requires code blocks to be *uniquely identifiable*. **Precedence**, denoted by $\rightarrow$, is a binary

```
1    /*** initialTokens($offset) ***/
2        $actorSymbol(token) = $val(initialOutputs, $offset);
3        $send(output)
4    /**/
5
6    /***fireBlock***/
7        $actorSymbol(token) = $get(input);
8        $send(output)
9    /**/
```

Figure 2.2: The C code template of `SampleDelay`.

relation that constrains a code block to appear somewhere before another. **Addition** is a binary operation that appends a code blocks directly after another. **Composition** is also a n-nary operation that combine multiple code blocks into one. Every code block has a set of variables used for this purpose. For instance, $c_1 \mid_v c_2$ is a composition of c_1 and c_2 at the variable v. In this case, we say the variable v in c_1 is bound to c_2. A general composition, $c_0 \mid_{v_1,...,v_n} c_1, ..., c_n$, may involve multiple variables, each of which is bound to a code block. This is also known as the **call-by-macro expansion**.

Code blocks can be extended with typing. Typing provides extra information about a code block. It can assist many generate-time analyses. Addition and composition can make use of the typing information to check their operands. For example, we can type a code block as either a *statement*, *expression*, or *neither*. Addition is allowed between two *statement*'s, or a *neither* and any type but it is disallowed between two *expression*'s. Composition enjoys similar benefits by extending the typing to code block variables and defining a custom set of typing rules. This typing information can be added statically in the code templates and inferred dynamically for composite code blocks.

Figure 2.2 illustrates the C code template for the SampleDelay actor. It consists of two

```
<CodeTemplate>  := <CodeBlock> | <CodeBlock><CodeTemplate>
<CodeBlock>     := /*** <BlockName> ***/ <Code> /**/ |
                   /*** <BlockName> ( <ParameterList> ) ***/ <Code> /**/
<BlockName>     := <Word>                                  // Any word character(s).
<ParameterList> := emptyList | <Parameter> | <ParameterList>, <Parameter>
<Parameter>     := $<Word>                                 // Any word prefixed with $.
<Code>          := [^/] | /[^*]+ | /*[^*]+ | /**[^/]+      // Any character sequence except
                                                           // the pattern /**/.
<Word>          := <Letter> <Word>                         // Any word character(s).
<Letter>        := // Any word character (i.e. [a-z]|[0-9]|_).
```

Figure 2.3: Grammar of the code template.

code blocks: `initialTokens` and `fireBlock`. `initialTokens` is parametrized by a parameter named `$offset`. Figure 2.3 shows the extended context-free grammar (in conjunction with regular expression) for parsing code templates. In general, a code block may have arbitrary number of arguments. Each argument is prefixed by the dollar sign "$" (e.g., $value, $width). Formally, the signature of a code block is the pair (N, p) where N is the name of the code block 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]

2.3.2 Adapters

An *adapter* is a key abstraction in the code generation framework. Every model component, such as an actor or director, is associated with an adapter. In our code generation framework, every adapter is defined in a Java class. There is a hierarchy of adapter classes which mirrors that

[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.

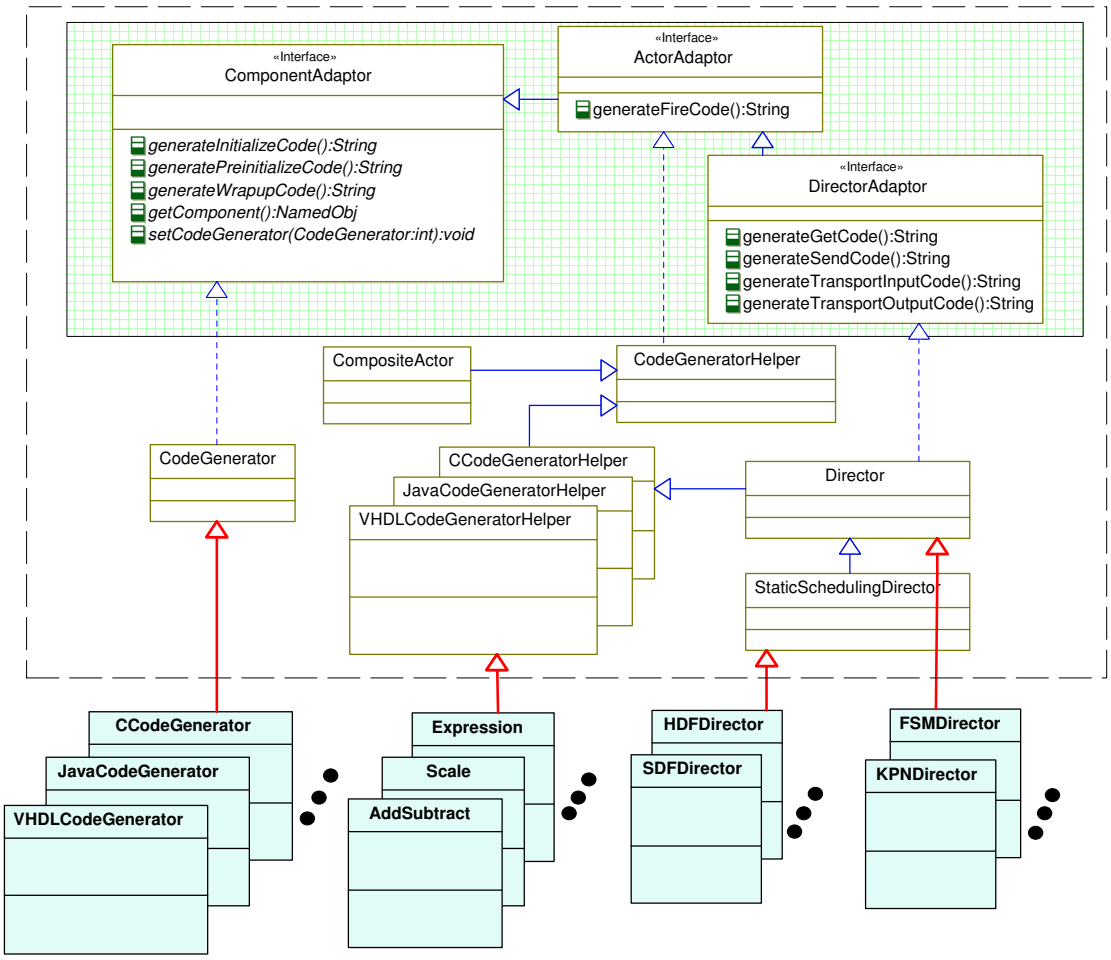

Figure 2.4: UML Class Diagram of the code generator software.

of the actor and director classes. Figure 2.4 is a UML diagram that shows the architecture of the adapter classes.

Adapters are active agents during generate time. They generate code for their model counter parts by instantiating code blocks and linking them together. There are two implicit assumptions behind the adapter abstraction. First, there exists a *uniform interface* for the adapters so that they can be composed together in a useful manner. Second, every adapter needs to be *identifi-*

able under a model scope. This ensures that a code generator can make use of the adapter. Under these assumptions, users can freely create extensions for new targets or replace an existing adapter with a new implementation.

We defined here the main abstractions provided in the code generation framework. The remainder of the report will show how to extend the code generation framework using these abstractions. Chapter 3 discusses the three main parts necessary in generating semantic-preserving code. Section 3.1 describes the abstraction of actor adapters and how actor parameters help in generate-time specialization. It also illustrates how to generate code for special actors such as Expression, CompositeActor, and higher-order actors. Director adapters are described in Section 3.2. In particular, the Kahn Process Network (KPN) is used as an exemplar to illustrate the extension of MoCs in the code generation framework. Chapter 3.3 illustrates how multiple datatypes can be extended. We discuss specifically how data polymorphism and coercion, or automatic type conversion, is resolved at generate time. Chapter 4 discuss how to retarget the framework for different programming interfaces. Finally, we will conclude by summarizing the contributions and pointing out potential future work.

Chapter 3

Semantics-Preserving Code Generation

3.1 Generating Code for Actors

Actors are basic building blocks in our modeling environment. Each actor has a set of ports and parameters. There are over three hundreds actors in the Ptolemy II library. Each actor has a unique semantics which defines the specific operations performed upon firing. Most actors are **atomic actors** whose behavior is defined by Ptolemy II framework developers. The definition is written in a Java class, thus implicit and unchangeable at the model level. On the other hand, **composite actors** are container of actors. Their behavior is specified using a sub-model. Model builders can model a component with complex behaviors by encapsulating multiple actors in a composite actor. In order to generate code, an actor requires a corresponding adapter. In the following section, we will present the interface of an actor adapter and show through examples how users can implement this interface for various actors. We will discuss the importance of actor parameters (Section sections 3.1.2) and naming (Section 3.1.3). We will then discuss implementing the non-trivial-case adapters for the Expression actor (Section 3.1.4), CompositeActor (Section 3.1.5), and higher-order

Methods	Descriptions
preinitialize	Topology or type checking prior to *initialize*.
initialize	Initialize the state before execution.
prefire	Check the preconditions for a firing.
fire	Fire the component.
postfire	Update the state after firing.
wrapup	Finalize the execution.

Table 3.1: The executable interface.

actors (Sections 3.1.6).

3.1.1 Actor Adapter Interface

The actor adapter interface consists of a set of *generate* methods that returns code for implementing the actor execution semantics. We call this the **actor code**, or code generated by an actor adapter. We implemented the adapter interface according to the Ptolemy abstract semantics. The essential parts in the abstract semantics are shown in Table 3.1. A more detailed description of the semantics can be found in [14]. The idea of the abstract semantics is to abstract the behavior of an actor. Concrete implementations can be then mapped to this interface. The abstract semantics is a well-established interface for actors. It is a common abstraction that encapsulates actor behavior. The generate methods of the adapter interface directly correspond to the abstract semantics. The adapter interface consists of the set of methods shown in Table 3.2.

Each of these generate methods performs certain generate-time analysis and return a code string as the result. Let us illustrate this using the AddSubtract actor. It has an output port and two input ports, *plus* and *minus*. Each of its input ports is a **multiport** that accepts multiple incoming connections (fan-ins). This is useful for performing n-nary operatoins without cascading. It provides

Method	Description
generatePreinitializeCode	return the *preinitializeCode*.
generateInitializeCode	return the *initializeCode*.
generatePrefireCode	return the *prefireCode*.
generateFireCode	return the *fireCode*.
generatePostfireCode	return the *postfireCode*.
generateWrapupCode	return the *wraupupCode*.

Table 3.2: The interface of actor adapters.

ease of use for modeling but raises the complexity of the actor behavior. Thus, this creates an opportunity for generate-time optimization. Figure 3.1 shows the code template we have defined for the AddSubtract actor.

To generate the actor fire code, the adapter first iterates through the connections to the `plus` and `minus` ports of the associated actor instance in the model. For each connection, it instantiates either the `firePlusInputCode` or `fireMinusInputCode` block which contains a code statement for performing the addition or subtraction. This is a simple loop-unrolling technique. The generated code is efficient for small number of input connections. However, it significantly increases the program code for cases where the number of input connections is large. We can imagine an alternate technique that combats this problem. It is to generate a loop that iterates through an array of channel variables at run-time. It allows reuse of the code for performing the arithmetic at the expense of introducing run-time overhead. Datatypes of the inputs may increase the amount of additional run-time overhead because the generated code may need to convert type dynamically. Given these two techniques, the adapter can determine at generate time which technique to use (e.g. by checking against a certain metrics, such as the number of different datatypes and input connections). Moreover, each of the generate methods in the adapter interface is to perform generate-time

```
1  /***preinitializeCode($type)***/
2    $type $actorSymbol(result);
3  /**/
4
5  /***initializeCode($type1, $type2)***/
6    $actorSymbol(result) = $convert_$type1_$type2($get(plus, 0));
7  /**/
8
9  /***firePlusInputCode($channel, $type1, $type2)***/
10   $actorSymbol(result) = $add_$type1_$type2($actorSymbol(result),
11                                             $get(plus, $channel));
12 /**/
13
14 /***fireMinusInputCode($channel, $type1, $type2)***/
15   $actorSymbol(result) = $subtract_$type1_$type2($actorSymbol(result),
16                                                  $get(minus, $channel));
17 /**/
18
19 /***fireOutputCode***/
20     $send(output, 0, $actorSymbol(result))    // semicolon is omitted for $send.
21 /**/
```

Figure 3.1: The AddSubtract code template.

analyses of this kind.

In addition to providing local optimization, the generate methods are conceptual divisions of the generated code that implements an actor. For example, all variable declarations are generated in the *generatePreinitializeCode*, and parameter values are assigned, or initialized, in *generateInitializeCode*. In most cases, the fire code contains the core functions of the actor.

Another important part of the adapter interface is generating the communication code that interacts with the environment, or other actor code. The *inter-actor code* is generated by the director adapter (discuss later in Chapter 3.2). This is realized by the addition of the `$get` and `$send` macros. These macros can be used in any code block instantiated by an actor adapter. Let us briefly introduce the semantics of these macros. The `$get` macro takes as arguments the name of an input port and a channel number. Its substitution, in the generated program, is an *expression* that represents the data token from that port channel. The `$send` macro takes as arguments the name of an output port, a channel number, and an expression of the data token to send. Its substitution is a *statement* that sends the given data token expression to the port channel. We will see in later chapters how they are used to generate code that passes data between multiple pieces of actor code.

3.1.2 Parameters

Actor parameters play a vital role in generate-time analyses. If a parameter is determined to be a fixed value or a small set or range of values, adapters can take advantages of this information to specialize the generated code for an actor. In a Ptolemy II model, it is possible to statically determine whether or not a parameter is constant [34]. When a parameter value is unconstrained, the adapter will need to generate a more dynamic implementation for the actor. Treating parameters as constants allows us to specialize the code for a subset of actor instances in the model.

3.1.3 Naming

A naming system is essential in code generation. It automates the generation of variable names in the program. This requires systematic naming for model components such that there is a unique string identifier for each object. Ptolemy II provides such a system. It has a notion of *named object*, each of which has a full name that uniquely identifies the object in the model. Actors, parameters, ports, and directors are subclasses of named objects. Our framework assumes this naming system and take advantage of it.

Naming is *abstracted* from code templates and *substituted* by the adapter at generate time. Since code templates is written for an entire class of components, a code block is used to generate code for multiple instances of the same class. The idea is to avoid explicit name references. A code block writer uses *relative* naming through a special macro called `$actorSymbol`. The line of code below declares a variable relative to a component instance:

```
int $actorSymbol(input);
```

The `$actorSymbol` macro is substituted at generate time with the full name of the component instance. If there exist two instances of this class, two declaration statements of different variable names are generated. E.g.

```
int Model_Expression_input;
int Model_Expression2_input;
```

3.1.4 Expression

Expression is a highly configurable actor. It has a parameter named *expression* that configures the behavior of the Expression actor. Users can use an expression language to set the parameter value. The Expression adapter packages a code generator for the expression language. Figure 3.3 illustrates the process of expression code generation. The adapter first parses the expression value

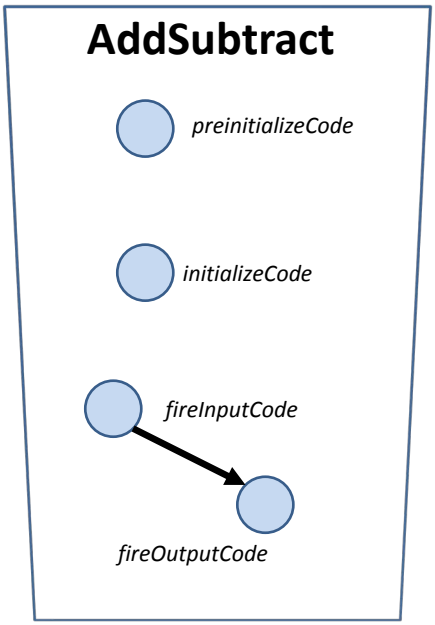

Figure 3.2: Partial ordering of the `AddSubtract` code blocks.

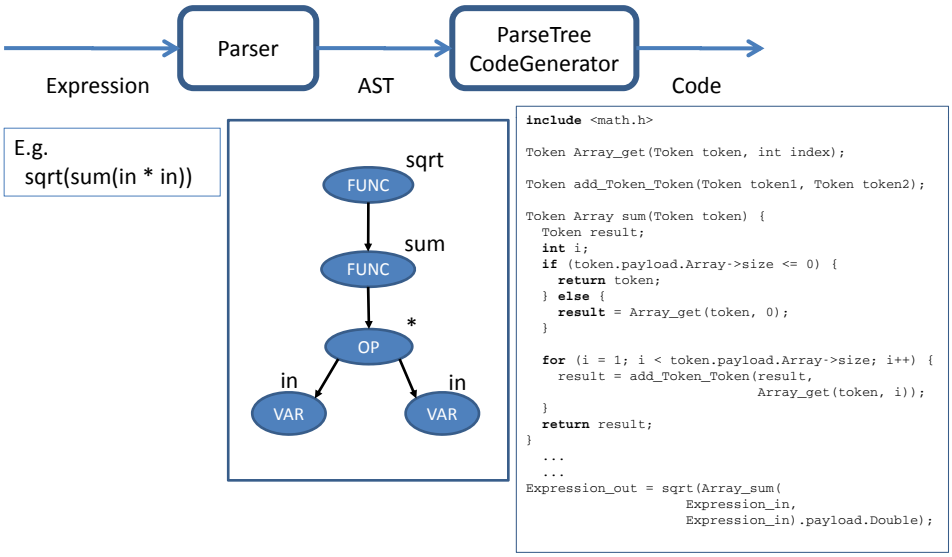

Figure 3.3: The `Expression` adapter.

into an abstract syntax trees (AST). It then generates code by traversing the given AST. The technique is well studied in compiler research, so we will leave the specific details outside of this report. Interested readers are referred to [16] for further details of the Ptolemy II expression language.

3.1.5 CompositeActor

A *CompositeActor* is an aggregation of a group of actors. It enables hierarchical composition and acts as a container for directors and actors. The BodyModel actor in Figure 1.1 is a CompositeActor. Each CompositeActor can be refined by a sub-model which enables designers to model complex behaviors through modeling. A CompositeActor works closely together with the model directors to execute the model. A CompositeActor has a notion of (outside) *executive* director and *inside* director. The executive director is the closest director that is outside of the CompositeActor, while the inside director is in the sub-model. Having an inside director is optional. In the absence of the inside director, the executive director is treated as the insider director. The CompositeActor is said to be **transparent**, meaning the CompositeActor acts merely as a visual container for its sub-components. Replacing a transparent CompositeActor with its sub-components make no differences in the semantics of the model. In the case where an insider director is present, the CompositeActor is said to be **opaque**. The CompositeActor serves as the boundary between MoCs.

The CompositeActor adapter is a key component in achieving code generation for mixing of MoCs, or sometimes called **heterogeneous modeling** [35]. It enables the composition of the code generation adapters. For example, the *generateFireCode* method of the CompositeActor adapter multiplexes the contained actors with various MoCs. It first transports the input data from outside by invoking the adapter of the executive and inside directors. It then delegates to the inside director adapter to specify the scheduling of the fire code of the contained actors. It finally transports the

produced data to the outside by invoking again the director adapters. The other *generate* methods of the CompositeActor adapter follow the same mechanism. The CompositeActor adapter glues together the actor and director adapter interfaces. Its interface provides a contract for mixing of MoCs. Newly defined MoC adapters can be added and compose with other MoC adapters without defining custom links.

3.1.6 Higher-Order Actors

Higher-order actors are actors that *operates* on actors. One of their usages is to provide ease of modeling without having users to copy multiple actor instances. This is the case for the `BodyModels` actor in Figure 1.1. `BodyModels` is an instance of the *MultiInstanceComposite* actor. It has an *instance* model (also shown in Figure 1.1) and a parameter named `nInstances` that indicates how many instances of the instance model to replicate. Moreover, Ptolemy II has a *PtalonActor* that generalizes higher-order composition. It uses a composition language for specifying large complex actor models [36].

To generate the proper code for these actors requires no extra support from the code generator. The reason is because these actors are not passed directly to the code generator. They are expanded, or replaced by regular actors, prior to generate time. We assume that the PtalonActor and MultiInstanceComposite are absent from run-time mutation. This ensures that the model structure is static. We can check this by performing a parameter analysis on the *nInstance* parameter of the MultiInstanceComposite and the Ptalon code contained by the PtalonActor to see if they can change at run time. Currently, we reject the model for code generation if run-time mutation can occur for these actors.

In addition to composing regular actors, other higher-order actors may introduce new

behavior of their own. The Case and ThreadedComposite actors are in this category. The Case actor is parametrized by a set of *case models*, or named refinement models. Its run-time behavior is driven by input data. It has a `control` port that receives input tokens. Upon firing, the value at the control signal is compared against the names of case models. The matched case model is then activated for firing.

A Case adapter is defined to generate code for the Case actor. We will intuitively describe the adapter implementation as follow. Its generatePreinitializeCode method generates a variable for the control signal and a set of symbolic values for the case model names. The generateFireCode method generates a switch statement which consists of a set of case statements. Each case statement is assigned a symbolic value generated previously for the case name. The case statement also contains the fire code generated for one of the case models. The case statement is executed upon matching the value of the control variable with its assigned the name value.

A more elaborate case is the ThreadedComposite [37]. It is an actor that executes a group of (contained) actors in a separate thread. It couples execution strategy with the actor semantics. We can generate code for the ThreadedComposite actor using strategy similar to the Case actor by defining a custom adapter. The idea is to map thread-specific code to the adapter interface. The ThreadedComposite adapter is not yet implemented but it can be realized within this framework.

3.2 Componentizing Models of Computation

As we mentioned earlier, a model director in Ptolemy II governs the interaction and any resources shared between actors. The two main operations are scheduling and communication. The director adapter is the corresponding code generation component which generates the scheduling and communication code. In addition, it controls the weaving of **actor code**, which is the code

Methods	Descriptions
generateGetCode	return a string that replaces the `$get()` macro.
generateSendCode	return a string that replaces the `$send()` macro.
generateTransportInputCode	return code that consume input at MoC boundaries.
generateTransportOutputCode	return code that send output at MoC boundaries.

Table 3.3: The interface of director adapters.

generated by the actor adapters. Correspondingly, let us call the code generated by the director adapter the **director code**. In this section, we will first present the interface of a director adapter. We will then discuss how to generate director code that implements concrete MoC semantics and show an example through Kahn Process Network (KPN) code generation.

3.2.1 Director Adapter Interface

Like the actor adapter, a director adapter interface has a set of generate methods that correspond to the executable interface (shown in Table 3.1. In addition, there are four extra methods that componentize the MoC-specific communication code. Table 3.3 shows a complete list of these methods: The `generateGetCode` and `generateSendCode` methods are invoked to substitute the `$get` and `$send` macros in the actor code. This is one place where actor adapters pass information to a director adapter. There is a well-defined boundary between actor code and director code. The `generateTransportInputCode` and `generateTransportOutputCode` methods generates code for communication that happens at the boundary between two directors. It is a special case of generating the get and send code.

3.2.2 KPN Code Generation

Kahn Process Network (KPN) is a model of computation that models the execution of concurrent processes. Under the KPN semantics, each actor in the model has a separate thread

of execution. Actors communicate through FIFO queues. Each connection represents an implicit queue that buffers data tokens. Actors can operate asynchronously in the abundance of input tokens. In the case where input tokens are not available, an actor blocks until the upstream actor produces the needed tokens. In the original formulation, these buffers are *infinite* size [38]. Parks later proposes in his Ph.D thesis a strategy to execute KPN models using bound size buffers when possible [39]. Our current implementation supports static allocation of buffers. User can specify individual buffer sizes by annotating the model.

A KPN program is *Turing-complete* [38]. Questions like termination or queue size bounds are undecideable. However, KPN does guarantee *determinacy* of data produced at the outputs of every actor. The sequence of data produced is independent from the scheduling of actor execution. This gives KPN a competitive advantage in addition to its highly concurrent nature. It makes a desirable programming model for parallel platforms. In a distributed setting, blocking read is a sufficient condition to ensure determinacy. We will also show an algorithm to reduce the amount of locking in a multi-threaded implementation. The remainder of this section will show an extension of the code generation framework to generate code that faithfully preserves the semantics of KPN.

Because of the modular design of the code generation framework, the extension requires only the addition of the `PNDirector` adapter and code template. They help generate the KPN-specific scheduling and communication code. Our example assumes the Pthread target. Table 3.4 lists the KPN-specific functions and data structures that are necessary in the program. We will show how each of them is generated by the `PNDirector` adapter.

The rightmost column of Table 3.4 shows the mapping of our design to the adapter interface we defined in Section 3.2.1. Each item in the middle column shows a section of code needed

MoC-specific code	KPN	Adapter interface
A. Scheduling code	1. Declaration of actor threads	*preinitializeCode*
	2. Declaration of synchronization variables	*preinitializeCode*
	3. Deadlock detection code	*preinitializeCode*
	4. Instantiation of actor threads	*initializeCode*
	5. Instantiation of synchronization variables	*initializeCode*
	6. Starting and joining of actor threads	*fireCode*
B. Communication code	1. Definition of buffer data structure	*preinitializeCode*
	2. Instantiation of buffers	*initializeCode*
	3. Implementation of `$get()`	*getCode*
	4. Implementation of `$send()`	*sendCode*
	5. Transport data between MoC boundaries	*transportCode*

Table 3.4: List of KPN-specific code in a Pthread program.

to implement the KPN semantics. In this case, Pthreads are used to provide concurrent execution for actor processes. Buffer data structures and synchronization variables together are used for coordinating the data exchange between threads. Each of these code sections are specified using one or several code blocks. The declaration of the buffer data structure is defined in a code block as shown in Figure 3.4: Each KPN buffer is a circular buffer whose capacity is statically assigned. It is equipped with a `readOffset` and a `writeOffset` variables to keep track of the next position to read and write. The size of the buffer cannot be determined by the difference between `readOffset` and `writeOffset` because it is equal to zero when the buffer is full and when it is empty. One solution is to sacrifice a buffer space, leaving it always unused, so full can be distinguished from empty. This design does not scale to arbitrary datatype (i.e. bad for data tokens of large size). Plus, data buffers can never be of size one, which is a special case that often allows optimization. The alternative is to use extra book-keeping variables to help determine the buffer size. We can try using an extra *count* variable whose value is updated after every read or write. However, every read and write would require locking because this variable is shared by both the reader and

```
1  /***KPNBufferDeclaration***/
2      struct KPNBuffer {
3          Token* data;                      // Data queue.
4
5          // Using a separate write count and read count,
6          // we can avoid using any semaphores. To determine
7          // the number of items in the buffer, we simply
8          // do (writeCount - readCount). Unsigned is necessary
9          // for working with wrap-around.
10         unsigned int writeCount;          // Number of items written.
11         unsigned int readCount;           // Number of items retrieved.
12         int readOffset;                   // Next index to read.
13         int writeOffset;                  // Next index to write.
14         int capacity;                     // Capacity of the buffer
15
16         // Synchronization variables.
17         pthread_cond_t* waitCondition;    // Condition variable for the buffer.
18         pthread_mutex_t* waitMutex;       // Buffer mutex.
19     };
20 /**/
```

Figure 3.4: Data structure of the KPN buffer.

writer. Our implementation is to have two count variables, `readCount` and `writeCount`, each of which is accessed by only the reader or the writer. To obtain the size of the buffer, we simply take (`writeCount` - `readCount`). As we will see, this helps to reduce locking in our read/write algorithm while ensuring consistency.

Scheduling is a main contributor to execution efficiency if processes share execution units. For Pthread code generation, a program has little control over the scheduling policies. Thus, we are handing off the scheduling responsibilities to the operation system. Since Pthread is a user-level thread package, users has the freedom to specify the mapping between user and kernel threads. Depending on the underlying platforms, this mapping may also influence the execution efficiency. Multiple threads that are mapped to a single kernel thread are called *unbound threads*, while a *bound thread* is one that is mapped to its own kernel thread (one-to-one). A Pthread has a schedul-

ing attribute, called `contentionscope`, that can be set to either `PTHREAD_SCOPE_PROCESS` (unbound) or `PTHREAD_SCOPE_SYSTEM` (bound). There are known tradeoffs between the two mapping models. Our code generator currently sets all actors threads to be unbound, except for actors that performs I/O operations.

The core of our KPN code generation is the implementation of data buffers. It is a main factor that determines the execution efficiency of the generated program. We focus on tackling the problem of *over-synchronization*. We will describe an algorithm that avoids locking in the common case and only performs locking for special cases (i.e. when buffer is full or empty). The key assumption is that every KPN buffer has only one reader and one writer. This is because every connection is made between an input port and an output port. Fanouts are considered multiple connections, one for each fanout degree. Based on this single-reader-single-writer assumption, a reader can safely consume tokens from a queue without locking, and likewise for a writer to produce tokens. Figure 3.5 and 3.6 shows the pseudo code of our algorithm.

The read operation blocks when the buffer is empty, and write blocks when the buffer is full. The two operations are almost identical except for the buffer condition they check for. In the common case where buffer is neither full nor empty, these operations can execute asynchronously. There are two critical sections (line 4-10 and 17) in each of the operation. The first (line 4-10) is for blocking the current executing thread when the corresponding block condition is true. The second critical section (line 17) prevents the signaling from racing with the other thread. The executing threads only enter these critical sections upon the *full* and *empty* conditions. This enables a KPN program to concurrently execute not only the component computation but also much of the communication code. Moreover, threads can utilize more efficiently the underlying parallelism provided

```
1  writeKPNBuffer(KPNBuffer b, Token data) {
2      if (isFull(b)) {                              // Check if buffer is full.
3          lock(b.waitMutex);                        // Acquire buffer-specific lock.
4          if (isFull(b)) {                          // Check again after locking.
5              incrementWriteBlockingThreads();      // If it is full, then wait.
6              wait(b.waitCondition, b.waitMutex);
7              if (isGlobalDeadlock()) {             // Check for a global deadlock.
8                  exit();
9              }
10         }
11         unlock(b.waitMutex);                      // Release lock.
12     }
13     b.data[b.writeOffset++] = data;               // Put token into buffer.
14     b.writeCount++;                               // Increment count.
15     if (isEmpty(b)) {                             // Check if buffer is empty.
16         lock(b.waitMutex);                        // Acquire buffer-specific lock.
17         signal(b.waitCondition);                  // Signal any blocked thread.
18         unlock(b.waitMutex);                      // Release lock.
19     }
20 }
```

Figure 3.5: Algorithm for writing to a KPN buffer.

```
1  Token readKPNBuffer(KPNBuffer b) {
2      if (isEmpty(b)) {                             // Check if buffer is empty.
3          lock(b.waitMutex);                        // Acquire buffer-specific lock.
4          if (isEmpty(b)) {                         // Check again after locking.
5              incrementReadBlockingThreads();       // If buffer is empty, then wait.
6              wait(b.waitCondition, b.waitMutex);
7              if (isGlobalDeadlock()) {             // Check for a global deadlock.
8                  exit();
9              }
10         }
11         unlock(b.waitMutex);                      // Release lock.
12     }
13     Token data = b.data[b.readOffset++];          // Read the data.
14     b.readCount++;                                // Increment count.
15     if (isFull(b)) {                              // Check if buffer is full.
16         lock(b.waitMutex);                        // Acquire buffer-specific lock.
17         signal(b.waitCondition);                  // Signal any blocked thread.
18         unlock(b.waitMutex);                      // Release lock.
19     }
20     return data;                                  // Return the token.
21 }
```

Figure 3.6: Algorithm for reading from a KPN buffer.

by the platform.

3.2.3 Domain-Specific Actors

Domain-specific actors are special actors in a MoC. Their function is tied to a particular MoC. They may require additional support from the directors, which specially recognize these actors and handle them differently during execution. They are similar to primitive constructs or keywords in a programming language. There are several examples of such domain-specific actors in the Ptolemy II actor library. For example, the Synchronous/Reactive (SR) domain has an `Absent` actor that outputs the absent signal, which is meaningful only within the SR and SR-derived domains. Likewise, `VariableDelay`, `TimeGap`, `SingleEvent`, and `EventFilter` are domain-specific actors for the DiscreteEvent (DE) domain.

The adapters for domain-specific actors often require extension to the adapter interface. Let us take the `NondeterministicMerge` actor as an example. It is a domain-specific actor in the Kahn Process Network (KPN). It has an output and multi-channel input port (multiport). It produces a stream of outputs by non-deterministically merging all of its input streams. Randomly picking an input channel and reading tokens from it is not an acceptable implementation, since the reading operation may block while tokens are available at other channels. Therefore, to complement the blocking `get` operation listed in Figures 3.6, we need a non-blocking primitive called `peek`.

Implementing the `peek` operation is straight-forward. Peeking can be offered as a macro (i.e. `$peek`) to the NondeterministicMerge adapter. Given our previous definition of the KPN buffer in Figure 3.4, we can implement `peek` as shown in Figure 3.7. In addition, an extra lock and condition variable, say `NmergeLock` and `NmergeCondition`, are also necessary in the NondeterministicMerge's code template. The fire code of NondeterministicMerge first peeks

```
1 // Return false if the given buffer is empty, otherwise true.
2 boolean peekKPNBuffer(KPNBuffer b) {
3     return b.writeCount - b.readCount > 0;
4 }
```

Figure 3.7: Algorithm for *peeking* at a KPN buffer.

through the input channels that has available tokens. If the pool is not empty, it randomly selects one to consume token from. If the pool is empty, it acquires the `NmergeLock` lock, peeks again, and waits on the condition if all channels are still empty. It requires the assistance of the director adapter to locate the actors that are senders to the input of NondeterministicMerge and associate `NmergeLock` and `NmergeCondition` in their `send` operations.

In general, domain-specific primitives are used in a restricted way, otherwise it can undermine the semantics of the MoC. In this particular case, peeking is only allowed for the NondeterministicMerge actor and not any other actor. Imagine the case where an actor is allowed to output values based on the availability of input tokens (i.e. The output is a boolean value that indicates whether its input port has tokens). The sequence of output values of this actor is determined by the scheduling of the actor execution. It thus violates the premise of output determinacy in KPN. It is therefore the responsibility of the director adapters to specially check and verify at generate time the use of these domain-specific primitives.

3.2.4 Heterogeneous Modeling

Heterogeneous modeling is an innovative technique that proposes composition of multiple MoCs. It attempts to maximize both analyzability and expressiveness by providing finer granularity in the use of MoCs. The Globally-Asynchronous, Locally-Synchronous (GALS) efforts are one example of mixing MoCs. In my previous work [40], I demonstrated the utility of composing KPN

and Synchronous Dataflow (SDF). The composition of PN and SDF effectively controls the degree of concurrency while retaining determinacy and understandability in the SDF sub-components. The Gravitation model in Figure 1.1 is an example of this. Composition between other MoCs are also possible. Goderis et al. proposed a classification for MoCs and used it to analyze which combination of MoCs are possible [35]. There are large interests in exploring heterogeneous modeling and formalizing it as a general technique.

Code generation for heterogeneous models requires extra gluing code. The gluing code bridges across MoCs' boundaries. It is generated by director adapters. The `generateTransportInputCode` and `generateTransportOutputCode` interface methods are used for this purpose. `generateTransportInputCode` returns code that transfers data from the input ports of the container to the ports connected on the inside, while `generateTransportOutputCode` returns code that transfers data from the output ports of the container to the ports connected on the outside.

In the case of composing KPN and SDF (where KPN is at the top level and SDF are the sub models), the transport input code copies data from a KPN buffer to a SDF buffer. This is precisely the `$get` code generated by the KPN director adapter and the $send code of the SDF director adapter. The transport output code is similar except the copying is of the opposite direction. Generating the transport code requires cooperation of the inside and outside director adapters. Generating transport code is the key mechanism in code generation for heterogeneous models.

3.3 Supporting Multiple Data Types

Datatype is another important aspect in the model. It mades up part of the semantics of the computation. The semantics of a component varies with the context of its datatypes. For instance,

the same operation performed on two different datatypes may have very different meaning and, thus, involve very distinct computation. Consider the addition between integers and between sets. The former is a scalar operation and the second a set union. In Ptolemy II, many actors are *polymorphic*, which means that they can operate over multiple datatypes. In the Gravitation model (shown in Figure 1.1), AddSubtract, MultiplyDivide, Scale, Repeat, SampleDelay are data polymorphic actors. In particular, Repeat and SampleDelay can propagate any types of tokens from the input to output port. Scale can operate on any datatype that supports multiplication. Similarly, AddSubtract and MultiplyDivide operate on any datatype that supports addition, subtraction, multiplication, and division. In this Chapter, we will show how to define datatypes in our code generation framework and how they interface with actor adapters through polymorphic functions. In Section 3.3.3, we will show how to use polymorphic functions to solve the coercion problem in code generation.

3.3.1 Data Type Library

A datatype is a representation of data in the model. Types are static properties of model components such as ports and parameters. Types are specified or inferred through static analysis of the model and thus prior to generate time. Each datatype implicitly defines a set of allowable operations between data. These operations give a generalization over fine-grained computation that is performed. Their admissibility is checked by a type checker.

Many modeling environments contain a rich set of types. Ptolemy II, in particular, support a large and extensible type system [17]. Thus, the code generation framework is designed to provide a library interface for these datatypes. The implementation of a datatype in the code generation framework is called a *codegen type*. The framework systematically incorporates multiple datatypes and allows modular extensions for new codegen types. The framework provides a `$type` macro

Macros	Descriptions
`$new`	creates a new data token for this type.
`$delete`	the inverse operation of `$new`.
`$clone`	return a copy of the data.
`$zero`	return the additive identity for data of this type.
`$one`	return the multiplicative identity for data of this type.

Table 3.5: Macros supported by a codegen datatype.

to query the datatype of a particular port and parameter. It avoids having the actor code blocks to reference any explicit types. The abstraction helps actor adapters to generate code independent of datatypes by hiding the detailed implementation of a particular type. It serves as a contract between the actor adapters and datatypes.

Table 3.5 the core functions of a codegen datatype. These operations are provided as macros used in the actor code templates. Each codegen type is required to implement the definitions of these operations. Figure 3.8 shows the String codegen type, defined using a code template.

To extend the framework with a new type, we simply add a file that contains the definition of these operations. The code generation framework currently supports 10 codegen types which include Array, Boolean, Complex, Double, Int, Long, Matrix, Pointer, String, and UnsignedByte.

3.3.2 Polymorphic Functions

Orthogonal to the codegen type definitions are the definitions for *polymorphic functions*. They are functions that accept argument(s) of multiple datatypes. Polymorphic functions has a notion of **type arity**, which refers to the number of type arguments that refines the function. Usually, the type arguments are the types of the actual arguments to the function but this is not a necessary requirement. Most useful polymorphic functions are either unary or binary but they can be n-nary in general. For example, negate (inverse), print, and toString are common unary polymorphic

```
1  /***String_new***/
2  /* Make a new integer token from the given value. */
3  Token String_new(char* s) {
4      Token result;
5      result.type = TYPE_String;
6      result.payload.String = strdup(s);
7      return result;
8  }
9  /**/
10
11 /***String_delete***/
12 Token String_delete(Token token, ...) {
13     free(token.payload.String);
14     /* We need to return something here because all the methods are declared
15      * as returning a Token so we can use them in a table of functions.
16      */
17     return emptyToken;
18 }
19 /**/
20
21 /***String_clone***/
22 Token String_clone(Token thisToken, ...) {
23     return $new(String(thisToken.payload.String));
24 }
25 /**/
26
27 /***String_zero***/
28 Token String_zero(Token token, ...) {
29     return $new(String(""));
30 }
31 /**/
32
33 /***String_one***/
34     // String_one is not supported.
35 /**/
```

Figure 3.8: Code template for the *String* datatype.

functions. There are add, subtract, multiply, divide, convert, and equals in the binary case. Each of these functions are tailored against the type of its arguments. The implementation to generate is determined the particular function specified in the actor code and type information given from the model. This process involves a rendezvous of information between the codegen type library, actor code, and model type knowledge.

The codegen type library allows the extension of polymorphic functions. Each polymorphic function is defined in a separate file, such as `add.c` or `negate.c`. It contains a set of definitions that implement the function with concrete codegen types. Figure 3.9 shows a simplified version of `add` between the Boolean, Int, String codegen types. This design supports n-nary operations. Each concrete definition is specified in a separate code block. It is a specialized implementation of the function given the types of the arguments. The maximum number of code blocks in any one of these function code template is bounded by T^n, where T is the number of codegen types and n is the type arity of the function. Since the number of definitions increases exponentially with arity, or the number of type arguments, it is laborious to write the complete implementation, especially for function with high type arity. The framework allow partial implementation that contains a subset of the concrete definitions. The framework warns the user if code generation requires the missing definitions. This allows the user to grow a function code template incrementally.

Let's illustrate how these pieces come together through examining the Scale code template shown in Figure 3.10. The code template consists of two code blocks: `scaleOnLeft` and `scaleOnRight`. The adapter chooses one of them to generate depending on the commutativity of the multiplication. It outputs as result the product of the input value and the value of the factor parameter. In particular, the multiplication operation it invokes is a binary poly-

```
1  /*** add_Boolean_Boolean() ***/
2  boolean add_Boolean_Boolean(boolean a1, boolean a2) {
3      return a1 | a2;
4  }
5  /**/
6
7  /*** add_Boolean_Int() ***/
8  int add_Boolean_Int(boolean a1, int a2) {
9      return $add_Int_Boolean(a2, a1);
10 }
11 /**/
12
13 /*** add_Boolean_String() ***/
14 char* add_Boolean_String(boolean a1, char* a2) {
15     char* result = (char*) malloc(sizeof(char) * ((a1 ? 5 : 6) + strlen(a2)));
16     strcpy(result, a2);
17     strcat(result, (a1 ? "true" : "false"));
18     return result;
19 }
20 /**/
21
22 /*** add_Int_Boolean() ***/
23 int add_Int_Boolean(int a1, boolean a2) {
24     return a1 + (a2 ? 1 : 0);
25 }
26 /**/
27
28 /*** add_Int_Int() ***/
29 int add_Int_Int(int a1, int a2) {
30     return a1 + a2;
31 }
32 /**/
33
34 /*** add_Int_String() ***/
35 char* add_Int_String(int a1, char* a2) {
36     char* string = (char*) malloc(sizeof(char) * (12 + strlen(a2)));
37     sprintf((char*) string, "%d%s", a1, a2);
38     return string;
39 }
40 /**/
41
42 /*** add_String_Boolean() ***/
43 char* add_String_Boolean(char* a1, boolean a2) {
44     char* result = (char*) malloc(sizeof(char) * ((a2 ? 5 : 6) + strlen(a1)));
45     strcpy(result, a1);
46     strcat(result, (a2 ? "true" : "false");
47     return result;
48 }
49 /**/
50
51 /*** add_String_Int() ***/
52 char* add_String_Int(char* a1, int a2) {
53     char* string = (char*) malloc(sizeof(char) * (12 + strlen(a1)));
54     sprintf((char*) string, "%s%d", a1, a2);
55     return string;
56 }
57 /**/
58
59 /*** add_String_String() ***/
60 char* add_String_String(char* a1, char* a2) {
61     char* result = (char*) malloc(sizeof(char) * (1 + strlen(a1) + strlen(a2)));
62     strcpy(result, a1);
63     strcat(result, a2);
64     return result;
65 }
66 /**/
```

Figure 3.9: Code template for the *add* polymorphic function.

```
1 /***scaleOnLeft***/
2     $send(output, 0, $multiply_$type(input)_$type(factor)($get(input, 0), $val(factor)))
3 /**/
4
5 /***scaleOnRight***/
6     $send(output, 0, $multiply_$type(factor)_$type(input)($val(factor), $get(input, 0));
7 /**/
```

Figure 3.10: Code template of the Scale actor.

morphic function. The invocation, `$multiply_$type(input)_$type(factor)`, consists of three macros expressions. Two of them, $type(factor) and $type(input), query the datatypes of the factor parameter and the input port, respectively. At generate time, they are replaced with labels of concrete codegen types, say for example, Int and Array (of Ints). The invocation now becomes `$multiply_Int_Array`, which is a valid reference for one of the concrete definitions of `multiply`.

3.3.3 The Coercion Problem

Coercion is the implicit, or automatic, type conversion of data tokens. It is needed when there exists discrepancy between the type interfaces of components. Implicit conversion helps to decrease the amount of explicit type conversion, where conversion actors are linked between two components of different types. Having a large number of these conversion actors in the model is a burden in terms of readability. Coercion, on the other hand, centralizes the handling of datatypes. It provides the model with type transparency and relieves model designers from dealing with type conversion explicitly.

However, coercion presents a challenging design problem for code generation. First, the model does not directly provide clues about where coercion take places. For explicit type conversion, there is an explicit conversion actor. We can handle easily by implementing the conversion

actor adapter. For implicit conversion, the first problem is *detecting* the coercions at generate time. Then, a second design problem is generating the proper and efficient conversion code, given that there are many possible conversions between types.

Coercion occurs when data is transferred between components of different types. It can happen between the outgoing and receiving ports in a connection. It can also take place between a parameter and a port where the parameter value is sent as port data, such is the case for the SampleDelay actor. Detecting coercion is a problem of finding all such pairs of components where their types are different. At generate time, the framework keeps a table of the coercion information, where each entry is a tuple of the form: `<fromComponent, toComponent, fromType, toType>`. Detection of coercion between port pairs is fully automated by traversing each connection in the model and checking for disparate types. The coercion detection between a parameter and port, however, requires user intervention. It is up to the adapter writers to specify this.

Given the information from the detection phase, the next task is to generate the conversion code. Our solution is to add a polymorphic function called `convert` and leverage our datatype library infrastructure. The `convert` function has a type arity of two (the type to convert from and the type to convert to). The code template of `convert` consists of concrete definitions between each pair of datatypes. Each definition gives an opportunity to write efficient code that converts data of one type to another.

Chapter 4

Retargeting

Retargeting is a technique to generate code for different programming interfaces. It bridges the barriers posed by platforms, programming languages, or standards by decoupling the details of the concrete implementation from design models. Model designers are relieved from the burden of mixing platform details while designing algorithms. Retargeting makes design models applicable in various areas of usage domains and allow many communities to take advantage of the model-based design methodology.

To enable retargetability, our code generation framework tags each of the adapters and code templates with a specific target. Each adapter generates code for a specific target, and each code template contains code compatible to the programming interface of that target. Thus, retargetability is achieved through *multiple libraries* of adapters. These libraries may overlap in order to maximize reusability. The overlapping is organized using a **target hierarchy** such that each parent is a target that provides a more general programming interface than the children targets. Consider the case for the C, POSIX, and WIN32 interfaces. C, in this case, is the parent for POSIX and WIN32. However, the problem of deciding the relationship between targets is hard in general. This is left for future research. The importance of the target hierarchy is to allow multiple targets to share

adapters and code templates when they are compatible.

After describing the general retargeting mechanism, we will now discuss several application areas where retargeting is useful. The areas listed are by no means exhaustive. They come from ideas and studies throughout the course of developing our framework. In most of these cases, we prototyped concrete extensions to our framework as demonstrations.

4.1 Targeting Other Platforms

A main mission of our framework is to generate semantics-preserving code that is portable across hardware, operating system, or middleware platforms. They make up various *execution platforms*. A code generator that can generate code for multiple platforms is said to be **retargetable**. Retargeting is a technique used to combat the non-uniformity of programming interfaces. The reason is because each target platform is set up with its own stack of software, which includes device drivers, firmware, operating systems, virtual machines, or middleware. They together present the user a unique programming interface for each platform. Let's take file system interface as an example. The C standard I/O library, `stdio.h`, provides a common interface for a set of primitive file I/O operations. Most file-based systems implement this interface. However, other operations that are more fine-grained, such as directory manipulations, are left outside of `stdio.h` and thus not standardized. They are defined in libraries that are tied to the platforms. This motivates the need for a retargetable code generator that can configure the generation of code according to platform characteristics.

Let us use the `DirectoryList` actor to illustrate how retargeting is performed in our framework. The actor lists the contents, or file names, of a given directory path. Let us also take POSIX and WIN32 as our targets. Traversing contents of a directory on these targets requires

platform-specific library support. POSIX requires the functions defined in `dirent.h`, whereas, WIN32 requires support from `windows.h`. The POSIX and WIN32 DirectoryList adapters generate code assuming these library interfaces, respectively. The DirectoryListing code templates are written as shown in Figure 4.1 (POSIX) and Figure 4.2 (WIN32).

A practical use case of retargeting is to generate code for embedded devices. They are often small and limited in resources such as power, processing speed, memory storage. Thus, their driver libraries are specialized for efficiency and often do not implement standard interfaces. It is also hard to set up a general-purpose operating systems on top of these platforms, due to the fact that they do not have the sufficient storage to hold the code. Thus, the software on these platforms need to be specifically tailored.

There are currently a number of operating systems that targets specifically for embedded devices. [41] shows an unofficial list of these embedded operating systems. We worked particularly with a popular real-time kernel, called OpenRTOS, that is ported for several micro-controllers [42]. It features multitasking and support concurrency control through locks, semaphores, and synchronized queues. In Section 4.3, we will discuss the generation of parallel code for the OpenRTOS target as well as other parallel programming models.

Programming standards and interfaces are expressed through a particular *programming language*. For example, the `WIN32` and `POSIX` libraries are standards in C. Our framework extends retargeting to different programming languages. A language is treated as a target. We currently experimented with C, Java, VHDL, and XML-based languages. Each language is an entry point for programming a particular platform. It provides concrete syntax for how a program is specified and frames the accessibility of the underlying resources. There are over 2,500 programming languages

```
1  /***preinitBlock***/
2      // This template is written for the POSIX API.
3      DIR *dp;
4      struct dirent *ep;
5      struct stat statbuf;
6      Token $actorSymbol(outputArray);
7  /**/
8
9  /***initBlock***/
10     $actorSymbol(outputArray) = $new(Array(0,0));
11 /**/
12
13 /***fireBlock($filepath)***/
14     dp = opendir ("$filepath");
15     if (dp != NULL) {
16         while (ep = readdir (dp)) {
17
18                         if (stat(ep->d_name, &statbuf) == -1) {
19                 printf("%d\n.", errno);
20                         }
21
22             if ($val(listOnlyFiles) && !S_ISREG(statbuf.st_mode)) {
23                     // Exclude non-files.
24             } else if ($val(listOnlyDirectories) && !S_ISDIR(statbuf.st_mode)) {
25                     // Exclude non-directories.
26             } else {
27                 Array_insert($actorSymbol(outputArray), $new(String(ep->d_name)));
28             }
29         }
30         (void) closedir (dp);
31     }
32     $send(output, 0, $actorSymbol(outputArray))
33 /**/
34
35 /***wrapupBlock***/
36     Array_delete($actorSymbol(outputArray));
37 /**/
```

Figure 4.1: The C code template of `DirectoryListing` for POSIX.

```
1  /***preinitBlock***/
2      // This template is written for the Windows API.
3      WIN32_FIND_DATA descriptor;
4      LARGE_INTEGER filesize;
5
6      TCHAR szDir[MAX_PATH];
7      //size_t length_of_arg;
8      HANDLE hFind;
9      Token $actorSymbol(outputArray);
10 /**/
11
12 /***initBlock***/
13     hFind = INVALID_HANDLE_VALUE;
14     $actorSymbol(outputArray) = $new(Array(0,0));
15 /**/
16
17
18 /***fireBlock($filepath)***/
19     // Prepare string for use with FindFile functions.  First, copy the
20     // string to a buffer, then append '\*' to the directory name.
21     strcpy(szDir, "$filepath");
22     strcat(szDir, TEXT("\\*"));
23
24     // Find the first file in the directory.
25     hFind = FindFirstFile(szDir, &descriptor);
26
27     // List all the files in the directory with some info about them.
28     do {
29         if (~$val(listOnlyFiles) & descriptor.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
30             Array_insert($actorSymbol(outputArray), $new(String(descriptor.cFileName)));
31
32         } else if (~$val(listOnlyDirectories)) {
33             Array_insert($actorSymbol(outputArray), $new(String(descriptor.cFileName)));
34         }
35     } while (FindNextFile(hFind, &descriptor) != 0);
36
37     FindClose(hFind);
38 /**/
39
40 /***wrapupBlock***/
41     Array_delete($actorSymbol(outputArray));
42 /**/
```

Figure 4.2: The C code template of `DirectoryListing` for WIN32.

[43]. The set of actively used languages evolves but continues to be many. A retargetable code generator is a translator that communicates design models across different languages.

4.2 Hardware Simulation and Synthesis

It has long been realized that DSP applications are more intuitively described using visual signal flow graphs. Commercial tools like Xilinx System Generator (XSG) provide a library of Simulink blocks which can be used as primitives for building DSP hardware. The library consists of multiplexors, adders, registers, memories etc and each can be parameterized for implementation on Xilinx FPGA's. It allows designers to perform bit and cycle accurate simulations of their system and later translate the design to Xilinx compatible gateware. For translation XSG internally calls Xilinx Core-Gen, a software tool that allows generating high level components like multipliers and memories with desired specification to directly map onto lower level Xilinx primitives. This approach limits its use to Xilinx FPGAs. It is non-trivial to re-target a XSG translated design to another FPGA family or an Application Specific Integrated Circuit (ASIC). Other commercial tools like Synplify DSP also based on Simulink provide similar features but can translate designs into platform independent specifications.

All such systems use execution semantics closer to the Synchronous Reactive (SR) domain in Ptolemy II. This is a natural outcome of the fact that synchronous logic is best described by SR semantics when all clock domains in a design differ by integer multiples of a base frequency. Truly asynchronous boundaries can be modelled using Discrete Event (DE) semantics which is underlying semantics in hardware description languages like Verilog and VHDL. As designs grow larger and span many clock boundaries a designer may need more flexibility with the expressiveness and abstraction of the modeling semantics. For example, parts of the system which are synchronous

and are timing or resource critical may be better modelled using SR semantics, while asynchronous interfaces may need DE semantics. At the top level of the system and for components that do not need a lot of optimizations, higher level semantics like that of Synchronous Data Flow (SDF) is preferable. SDF semantics implicitly infer queues between actors to take care of differences in firing rates (clock speeds) between blocks. Translation from SDF to gateware is non-trivial and there has been previous work in attempting this using Ptolemy Classic [44]. Our current experiment extends the existing code-generation infrastructure, with VHDL as a new target, to generate behavorial Register Transfer Level (RTL) VHDL is independent of implementation platforms and can be targeted to ASICs as well. The details of the VHDL code generation target is documented in the technical report [45]. The introduction of VHDL code-generation in Ptolemy II helps set the stage for further research in the area of improving heterogeneous design environments for gateware and hardware designers.

4.3 Parallel and Distributed Computing

Parallel programming is the specification of concurrency. Its applications range from desktop software running on personal computers to climate modeling and astrophysics computing that are spread across thousands of supercomputing nodes [46, 47]. Model-based code generation is essential in parallel program design which often utilizes high-level abstractions of concurrency control, reaching beyond primitives like semaphores, mutexes, and monitors. However, finding the right abstractions is not an easy task, and it continues to be an active field of research. There have been various approaches experimenting with abstractions in both traditional programming languages and graphical modeling languages. Preserving *backward compatibility* is a major focus so users are not forced to learn a completely new language. Examples are language extensions such

as parallel Erlang [48] and OpenMP [49], which are extensions to the core languages. These extension are implemented using code annotations and compiler library support. In addition, other considerations for parallel design patterns are scalability, performance, compositionality, decidability, understandability and adoptability. Current research continues to improve the quantification of these measures.

A parallel program can be classified in three ways. First, **program parallelization** is described at the level of task and data. *Data parallelism* takes the form of breaking up program data into pieces and distributing them for uniform execution. The execution units are designed to perform the same operations. This form of parallelism relies heavily on the uniformity and structure of the data. It is generally considered fine-grained parallelism. *Task parallelism* takes advantage of dividing work into several functionally different tasks which can be execute concurrently. To minimize communication overhead, the boundary between tasks is drawn such that only small amount of data is shared while the majority of data are encapsulated within each individual task. **Communication architecture** is another important class of classification. Parallel programs relies on two main underlying architectures which are *message-passing* and *shared-memory*. Message-passing provides explicit control in sending and receiving data, in the form of messages, as well as delivery methods such as blocking and asynchronous messages. Shared-memory platforms on the other hand make the distinction between local and shared data and implicitly manages consistency of shared data. The platform is notified of data updates and can make decisions, using global knowledge, about how to efficiently perform the appropriate reads and writes. The third classification is **scheduling policy** which describes the mapping of functional tasks to execution units. *Static scheduling* minimizes the coordination overhead, while *dynamic scheduling* is more flexible and effective in

load-balancing.

Actor modeling in conjunction with concurrent models of computation provides an intuitive programming interface for specifying concurrent design patterns. The KPN model of computation, described in Section 3.2.2, in particular provides an intuitive syntax and semantics for understanding concurrency. It is also possible to transform into efficient code. Code generation for KPN serves as a starting point for us to use MoCs to experiment with parallel programming design. Using timed MoCs would further allow us to introduce time semantics into concurrency control. We have experimented with retargeting KPN to Pthread programming, as described in Section 3.2.2, as well as multitasking for a real-time embedded kernel, called OpenRTOS. One interesting target platform is the Message Passing Interface (MPI). It is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes [8]. It is portable and independent of programming languages. MPI belongs in layers 5 and higher of the OSI Reference Model with socket and TCP being used in the transport layer.

We extended the KPN code generation for the MPI target. In order to map actor execution to processing nodes, the code generator goes through two main stages: *partitioning* and *code generation*. First, we assume that the actors and connections are annotated with node and edge weights that correspond to the amount of computation and communication overhead each actor and connection represent. We can use these weights to influence decisions made in the partitioning stage. They also serve as parameters for us to later explore the partitioning design space. In the partitioning phase, we analyze the model and generate clustering information using the weights and the given

number of processors. It replaces the weight information with processor IDs, called *ranks*, and MPI communication buffer IDs. The code generation phase infers the necessary information from these partitioning and buffer annotations. It then generates a MPI program instance which is one particular implementation of the model based upon the given partitioning. Ideally, there would be a tuning phase that takes advantage of reconfiguring the partitioning parameters to create multiple program instances. It first executes the generated program and receives profiling information from each processors. These profiling statistics serve as feedback information to further adjust the edge and node weights. The tuning gives the partitioner a better estimate of its parameters, thus a better partition. This mechanism provides a way to systematically explore the design and implementation space. For further details about the work in MPI code generation, readers are referred to the technical report in [50].

4.4 Model Checking

Another useful application that can leverage the model-based code generation is formal verification through modeling checking. Our code generation framework is currently retargeted to Real-Time Maude, a specification language designed for checking real-time properties of a system. Model checking tools take as inputs a system model description and a set of constraints. The system model description is usually expressed in a custom language. The SPIN model checker, e.g., reads in system descriptions specified in the Process Meta Langauge (Promela). There are several other verification systems such as Maude and NuSMV, each invented its own specification language. The constraints on the other hand are written using Linear Temporal Logic (LTL) and/or Computation Tree Logic (CTL). Given these two inputs, a model checker attempts to either conclude the system is safe or find a counter-example where the system would violate the constraints. Automatic code

generation comes in handy for extracting the system description from a model. This form of re-targeting enable a smooth hand-off from design to verification. It also minimizes the manual labor required to iterate rapidly between the two design phases. By encapsulating the coupling between the design and verification languages in our code generator, we can effectively reuse the same tool in designing multiple systems.

Specifying concurrency is a central theme in verification. Using actor models, like those of Ptolemy II, that are defined with specific MoCs is advantageous. It makes possible for code generation to preserve the pre-defined communication and execution semantics in the model. The code generator can unambiguously transform a model for different verification tools to detect concurrency problems, such as livelocks, deadlocks, or race conditions. The notion of MoCs anchors a fixed interpretation on the model. Code generation bears the responsibility of translating this interpretation to source code with appropriate syntax. This approach of automatically generating system descriptions is used in several projects. Corbett et al. [51] demonstrated the extraction of finite-state models from Java source code, written with a subset of Java. The SLAM [52] verification engine, now used in Microsoft, automatically constructs *abstraction models* of a C program. Henzinger et al. developed a similar tool, called BLAST [53]. Both BLAST and SLAM operate through a series of automatic generation of abstraction models, while in each subsequent stage increasing the accuracy of the model. These tools are becoming widely successfully in formally verifying system protocols and architectures. Model checkers such as SPIN, Java PathFinder and SLAM [52] are being adopted by the industry as part of the engineering toolsuite and required in the standard development cycle.

Chapter 5

Conclusion

5.1 Summary

To summarize the main contributions of this report, we formulated model-based code generation as a graph problem. A framework, consisting of adapters and code templates, is designed to provide model designers reusable code generation components. The actor and director adapter interfaces are defined for semantics-preserving code generation. The adapter and code template abstractions can seen as design patterns for the code generation of Ptolemy II modeling language. The success of their usage is demonstrated with extension for various actors, MoCs, and datatypes. These patterns are also shown to be resilient for various target platforms.

We illustrated the adapters for several special actors, including Expression, Case, and CompositeActor. In addition, the CompositeActor adapter and director adapter interface support code generation for heterogeneous models. We particularly included a detailed illustration of the key mechanisms in generating code for Kahn Process Network (KPN) models. The framework is also adaptable to the extensible type system of Ptolemy II. It reuses the Ptolemy II type checker and supplement an extensible datatype library used in generate-time specialization.

We also explored retargeting for several applications. Retargeting for different languages

and platforms increases the portability of model execution. We come to realize that some models of computation are well-matched for particular application domains. For instance, we experimented with KPN to express parallel and distributed programs. We used SR models to simulate and synthesize hardware by generating VHDL code. We also demonstrated retargeting as a good tool for model checking where system description is automatically generated from a model.

5.2 Future Work

Future explorations can head toward several directions. Throughout the report, I described code generation as a monolithic process for a model. It is sometimes called *whole-system generation*. However, code generation can also be incremental. It helps to increase turn-around time for applications such as model debugging, system and legacy code integration. The next steps in this direction is to modularize code generation. E.g., researcher Stavros Tripakis is designing a set of interface theories for model components. One of the usages is to allow stand-alone compilation of sub-models. The generated code can be used in partial code simulation for quick debugging session. Alternatively, it can be fed back to the model to accelerate model execution.

A second important but challenging exploration is to verify this code generation approach. Verifying the entire code generation framework is a humongous, if not impossible, task because there is a large number of states for the code generator. Instead, we can start with verifying the generated program. One possible approach is to insert annotations, like pre- and post-conditions, into the generated program and have a program checker or theorem prover to check these conditions.

Another direction is to test the efficiency and performance of this model-based code generation approach. We can conduct future experiments to do the same designs using actor-oriented modeling and compare the results with hand-written programs. We can take measurements in terms

of the time to produce these designs, quality of the products, and flexibility to requirement changes or customization. There remains much to done to fully realize the capability of model-based code generation.

Bibliography

[1] Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002) Foreword By-Jacoboson, Ivar. 1

[2] Brayton, R.K., Sangiovanni-Vincentelli, A.L., McMullen, C.T., Hachtel, G.D.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, Norwell, MA, USA (1984) 2

[3] Keutzer, K.: Dagon: Technology binding and local optimization by dag matching. In: 25 years of DAC: Papers on Twenty-five years of electronic design automation, New York, NY, USA, ACM (1988) 617–624 2

[4] Zalewski, J., Kornecki, A.: Qualification of software development tools for airborne systems certification. In: Workshop on Software Certification Management, Institute of Electrical and Electronics Engineers 3

[5] Bock, C.: SysML and UML 2 support for activity modeling. Syst. Eng. **9**(2) (2006) 160–186 5

[6] Rumbaugh, J.: The unified modeling language reference manual, second edition. Journal of Object Technology **3**(10) (2004) 193–195 5

[7] OMG: System modeling language specification v1.1. Technical report, Object Management Group (2008) 5

[8] Hardebolle, C., Boulanger, F.: Modhel'x: A component-oriented approach to multi- formalism modeling. In: MODELS 2007 Workshop on Multi- Paradigm Modeling, Nashville, Tennessee, USA, Elsevier Science B.V. (2007) 5

[9] Pree, W., Templ, J.: Modeling with the timing definition language (tdl). In: Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services. LNCS, San Diego, CA, Springer (2006) 5

[10] Herrera, F., Villar, E.: A framework for embedded system specification under different models of computation in SystemC. In: Design Automation Conference (DAC), San Francisco, ACM (2006) 5

[11] Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Circuits and Systems **23**(1) (2004) 17–32 5

[12] Goessler, G., Sangiovanni-Vincentelli, A.: Compositional modeling in Metropolis. In: EMSOFT, Grenoble, France, Springer-Verlag (2002) 5

[13] Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE **91**(2) (2003) 127–144 5

[14] Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H., Brooks, C., Lee, E.A., Liu, X., Zhao, Y., Zheng, H., Bhattacharyya, S.S., Brooks, C., Cheong, E., Goel, M., Kienhuis, B., Lee, E.A., kit Leung, M., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J.,

Smyth, N., Tsay, J., Vogel, B., Williams, W., Xiong, Y., Zhao, Y., Zheng, H.: Heterogeneous concurrent modeling and design in java (volumes 1-3. Technical report (2004) 5, 25

[15] CHESS: The public website of the ptolemy project. http://ptolemy.berkeley.edu 5

[16] : The ptolemy expression language. http://ptolemy.eecs.berkeley.edu/~ptII/ptolemyII/ptII1.0/ptII1.0/doc/expression.htm 8, 31

[17] Xiong, Y.: An Extensible Type System for Component-Based Design. Ph.d. thesis (May 1 2002) 8, 11, 43

[18] Lee, E.A., Messerschmitt, D.G.: Synchronous data flow: Describing signal processing algorithm for parallel computation. In: COMPCON. (1987) 310–315 8

[19] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu/ 8

[20] Lee, E.A., Sangiovanni-vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (1998) 8

[21] Pino, J.L., Lee, E.A., Bhattacharyya, S.S.: A hierarchical multiprocessor scheduling system for dsp applications. Asilomar Conference on Signals, Systems and Computers **0** (1995) 122 9

[22] Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems **18** (1999) 742–760 9

[23] Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis (1993) Chair-Lee, Edward A. 9

[24] Zhou, G.: Dynamic dataflow modeling in ptolemy ii. Master's thesis, UC Berkeley (2004) 9

[25] Boussinot, F., de Simone, R.: The ESTEREL language. Proceedings of the IEEE **79**(9) (Sep 1991) 1293–1304 10

[26] Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the signal language and its semantics. Sci. Comput. Program. **16**(2) (1991) 103–149 10

[27] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE **79**(9) (Sep 1991) 1305–1320 10

[28] Zhou, Y., Lee, E.A.: Causality interfaces for actor networks. ACM Trans. Embed. Comput. Syst. **7**(3) (2008) 1–35 10

[29] Edwards, S.A.: The specification and execution of heterogeneous synchronous reactive systems. PhD thesis, Berkeley, CA, USA (1998) 10

[30] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj, Springer-Verlag (2001) 327–353 12

[31] Batory, D.: Feature-oriented programming and the ahead tool suite (2004) 12

[32] Zhou, G.: Partial evaluation for optimized compilation of actor-oriented models. PhD thesis, EECS Department, University of California, Berkeley (May 2008) 17

[33] Musser, D.R., Stepanov, A.A.: Generic programming. In: ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation, London, UK, Springer-Verlag (1989) 13–25 18

[34] Neuendorffer, S., Lee, E., Keutzer, K., Reader, S.: Automatic specialization of actor-oriented models in ptolemy ii (2002) 28

[35] Goderis, A., Brooks, C., Altintas, I., Lee, E.A., Goble, C.: Heterogeneous composition of models of computation. Technical Report UCB/EECS-2007-139, EECS Department, University of California, Berkeley (November 2007) 31, 42

[36] Cataldo, A.: The Power of Higher-Order Composition Languages in System Design. PhD thesis, University of California, Berkeley (December 2006) 32

[37] Lee, E.A.: Threadedcomposite: A mechanism for building concurrent and parallel ptolemy ii models. Technical Report UCB/EECS-2008-151, EECS Department, University of California, Berkeley (Dec 2008) 33

[38] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, J.L., ed.: Information Processing, Stockholm, Sweden, North Holland Publishing Company (August 1974) 471–475 35

[39] Parks, T.M.: Bounded scheduling of process networks. PhD thesis, Berkeley, CA, USA (1995) 35

[40] Leung, M.K., Lee, E.A.: An extensible software synthesis framework for heterogeneous actor models. In: SLA++P 2008, Model-driven High-level Programming of Embedded Systems, Artist (March 2008) See SLA++P 2008. 41

[41] Wikipedia: List of embedded operating systems. http://en.wikipedia.org/wiki/List_of_operating_systems#Embedded 52

[42] The FreeRTOS.org Project: Openrtos. http://www.freertos.org 52

[43] Kinnersley, B.: The list of computer languages. http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm 55

[44] Williamson, M.C.: Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. Technical Report UCB/ERL M98/45, University of California, Berkeley (June 1998) 56

[45] Leung, M.K., Filiba, T.E., Nagpal, V.: Vhdl code generation in the ptolemy ii environment. Technical Report UCB/EECS-2008-140, EECS Department, University of California, Berkeley (Oct 2008) 56

[46] W. M. Washington, J. W. Weatherly, G.A.M.A.J.S.J.T.W.B.A.P.C.W.G.S.J.J.A.V.B.W.R.J., Zhang, Y.: Parallel climate model (pcm) control and transient simulations. Climate Dynamics (2000) 56

[47] Makino, J. Fukushige, T.K.M.N.K.: Grape-6: Massively-parallel special-purpose computer for astrophysical particle simulations. PUBLICATIONS- ASTRONOMICAL SOCIETY OF JAPAN (2003) 56

[48] Armstrong, J., Virding, R., Wikstrm, C., Williams, M.: Concurrent programming in erlang (1993) 57

[49] Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE **5**(1) (1998) 46–55 57

[50] Leung, M.K., Liu, I., Zou, J.: Code generation for process network models onto parallel architectures. Technical Report UCB/EECS-2008-139, EECS Department, University of California, Berkeley (Oct 2008) 59

[51] Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: In Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 439–448 60

[52] Ball, T., Rajamani, S.K.: The slam toolkit. In: CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2001) 260–264 60

[53] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Not. **37**(1) (2002) 58–70 60