Partial Evaluation for Optimized Compilation of Actor-Oriented Models

Gang Zhou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-53
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-53.html

May 16, 2008

Partial Evaluation for Optimized Compilation of Actor-Oriented Models

by

Gang Zhou

B.S. (Peking University) 1995
M.S. (University of California, Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Spring 2008

The dissertation of Gang Zhou is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2008

Partial Evaluation for Optimized Compilation of Actor-Oriented Models

Abstract

Partial Evaluation for Optimized Compilation of Actor-Oriented Models

by

Gang Zhou

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

One major achievement of traditional computer science is systematically abstracting away the physical world. The Von Neumann model provides a universal abstraction for sequential computation. The concept is so simple and powerful for transformational systems (vs. reactive systems [Berry1989]) that any traditional program can run regardless of underlying platform —- whether it is a supercomputer or a desktop. Embedded software systems, however, engage the physical world. Time, concurrency, liveness, continuums and reactivity must be remarried to computation [Lee2006]. In order to reason about these properties, a programming model must be able to express these properties directly.

The traditional techniques for doing concurrent programming on top of sequential programming languages like C use threads complemented with synchronization

mechanisms like semaphores and mutual exclusion locks. These methods are at best retrofits to the fundamentally sequential formalism and are hard to understand, difficult to debug, brittle and error-prone.

Actor-oriented design presents a high level abstraction for composing concurrent components. There is a rich variety of concurrency formalisms that span the whole spectrum from most expressive to most analyzable. In particular, I will focus on one particular model of computation called heterochronous dataflow (HDF) which strikes a nice balance between expressiveness and analyzability.

However, high level abstraction often introduces overhead and results in slower systems. In component based design, generic components are highly parameterized for reusability and thus are less efficient. The well-defined interface between components results in flexible composition but increases the cost of inter-component communication through the interface.

In the second part of this thesis, I will address the problem of generating an efficient implementation from a high level specification. I use partial evaluation as an optimized compilation technique for actor-oriented models. Compared with traditional compiler optimization, partial evaluation for embedded software works at the component level and heavily leverages domain-specific knowledge. Through model analysis—the counterpart of binding-time analysis in the use of partial evaluation for general purpose software [Jones1993], the tool can discover the static parts in the model including data types, buffer sizes, parameter values, model structures and

execution schedules, etc. and then exploit the already known information to reach a very efficient implementation. I use a helper-based mechanism, which leads to flexible and extensible code generation framework. The end result is that the benefit offered by high level abstraction comes with (almost) no performance penalty.

The code generation framework described in this dissertation has been released in open source as part of Ptolemy II and can be downloaded from http://ptolemy.org/.

Professor Edward A. Lee
Dissertation Committee Chair

To my wife Yan, my son Brian

and my parents Jiangu Zhou and Shiying Feng

Contents

List of Figures

Acknowledgments

I am deeply grateful to my research advisor, Professor Edward Lee. He led me into this research area and supported me during my most difficult time. This thesis would never have been written without his guidance and encouragement. I would also like to thank him for the pleasant group atmosphere and outstanding research environment he created. I am deeply inspired by his enthusiasm and devotion to pursuing excellent research.

I am also very grateful to Professor Jan Rabaey, Professor David Auslander and Professor Robert Brayton for serving on my qualifying exam and dissertation committee, and for providing valuable advice to my research as well as my career.

I have enjoyed working in the Ptolemy group and it has been a great pleasure to work with Adam Cataldo, Elaine Cheong, Thomas Huining Feng, Man-kit Leung, Xiaojun Liu, Eleftherios Matsikoudis, Haiyang Zheng, Yang Zhao, and Ye Zhou. In particular, Man-kit Leung and Ye Zhou collaborated on this research project from day one and I am deeply appreciative of their work. I would also like to thank Christopher Brooks for his excellent management of the Ptolemy project and always swift response to my questions, and Mary Stewart for the numerous help she provided. Thanks to Ruth Gjerde for being such a helpful graduate assistant. I am so fortunate that she's always there to help me overcome many hurdles I encountered during my years at Berkeley.

Lastly, but most importantly, I thank my family: my wife Yan Wang, my son

Brian Tianxiang Zhou, my father Jiangu Zhou, my mother Shiying Feng, and my brother Zhi Zhou. I am extremely grateful for Yan's support over the years. Brian's laughter make my life full of joy. My parents have supported me all my life and always believe in me. I dedicate this dissertation to them.

Chapter 1

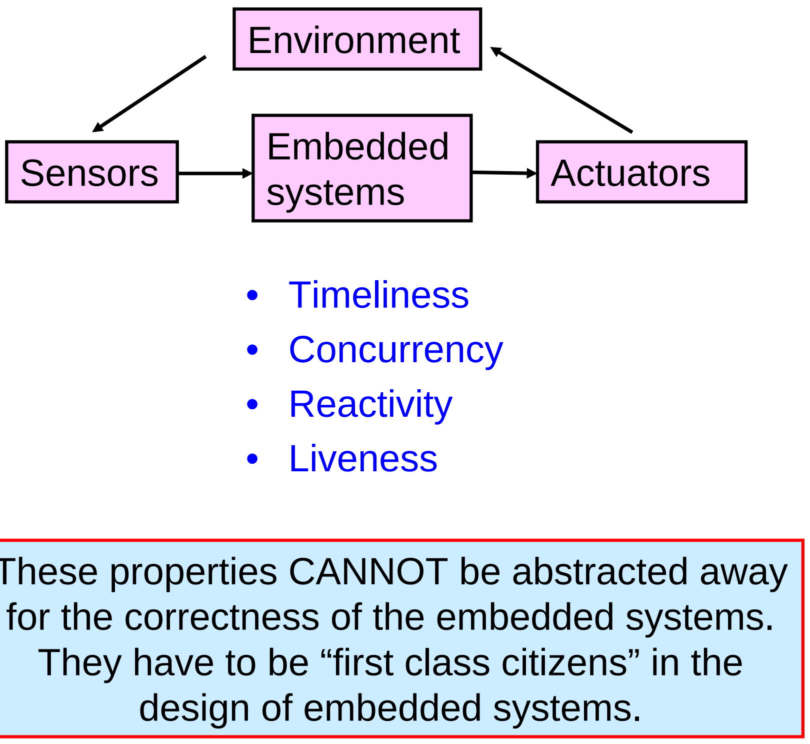
Introduction

The first operational, electronic, general purpose computer called ENIAC (Electronic Numerical Integrator and Calculator) was built during World War II for computing artillery tables. Ever since then computers have become faster and more powerful, due to technology (Moore's law) and architecture (caches and pipelining, etc.) advancements on the hardware side. On the software side, high level programming languages (C++, Java, among the most popular) were designed over the years to improve programmers' productivity. These inventions are fundamentally rooted in the abstraction for sequential computation. That is, computation is a series of data transformations, which can be done in a faster and more efficient way with the inventions of new computer architectures, new algorithms, new language features. One only needs to make sure the end result of the computation remains the same. So, in this sense, general purpose computing still tries to solve the same problem the

ENIAC was built for. However, as the technology progresses, we are not satisfied with just computing artillery tables; instead, we are much more interested in automatically controlling the whole process of artillery firing in real-time. In this case, the computer system needs to engage the physical world and we arrive at the world of embedded computing.

Embedded software has been traditionally viewed as software running on small computers with scarce resources. Therefore it is written with assembly language to maximize efficiency and predictability. Programming languages like C are also used to improve productivity and portability. These imperative programming languages abstract how a Von Neumann computer operates in a sequential manner. They are good matches for general-purpose software applications, which are essentially a series of data transformations. However, in the embedded world, the computing system constantly engages the physical system. Thus the physical system becomes an integral part of the design and the software must operate concurrently with the physical system.

The basic techniques for doing concurrent programming on top of traditional programming languages like C and Java use threads (for C there are various thread libraries such as pthreads; for Java threading is built into the language), complemented with synchronization mechanisms like semaphores and mutual exclusion locks. These methods are at best retrofits to the original fundamentally sequential formalism. Therefore they are difficult to reason about and guarantee correctness [Lee2006].



These properties CANNOT be abstracted away for the correctness of the embedded systems. They have to be “first class citizens” in the design of embedded systems.

Figure 1.1: Characteristics of embedded systems.

In fact, according to a survey [WDF] conducted by the Microsoft Windows Driver Foundation team, the top reason for driver crashes is concurrency and race conditions. This is not acceptable for embedded applications that are real-time and often safety-critical. We need alternative concurrency formalisms to match the abstractions with the embedded applications.

Besides concurrency there are other equally important issues a designer must consider when designing embedded systems (see Fig. 1.1). Unlike general-purpose computing where improving average performance is the key, embedded systems usually must guarantee the timeliness each time they execute. Unlike general-purpose computing where non-terminating programs are considered defective programs, embedded systems continuously interact with the physical processes through sensors and actuators and thus liveness of programs must be assured. Unlike general-purpose computing which can be abstracted as pure data transformations, embedded systems react at their environment's speed and thus have real-time constraints; and embedded systems usually react to multiple sources of stimuli and thus are concurrent. As one can see, timeliness, concurrency, liveness and reactivity are all related. These properties cannot be abstracted away for the correctness of the embedded systems. They have to be"first class citizens" in the design of embedded systems. In another words, programming models for embedded systems must be able to directly express these properties if they want to reason about these properties. Retrofitting with external means such as priority and semaphores will only lead to designs that are hard to understand, difficult to debug, brittle and error-prone.

The Ptolemy group has been advocating actor-oriented design as the programming model for the embedded systems. The actor model was originally proposed by Hewitt to describe the concept of autonomous reasoning agents [Hewitt1977] and extended by Agha and others to describe a formalized model of concurrency [Agha1986]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. Lee evolved the term to embrace a larger family of models of concurrency that are often more constrained than general message passing. Lee's

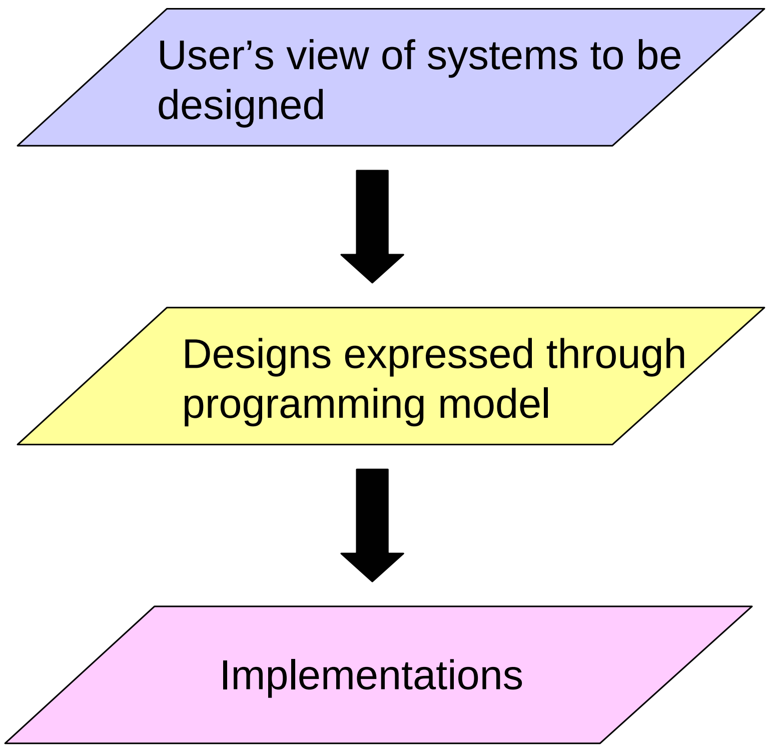

User's view of systems to be designed
Designs expressed through programming model
Implementations

Figure 1.2: A simplified view of design process.

actors are still conceptually concurrent, but unlike Agha's actors, they need not have their own thread of control, and communication is still through message passing but need not be strictly asynchronous.

My thesis can be separated into two parts. The first part corresponds to the top arrow in Fig. 1.2. It's about expressing designs through programming model with appropriate concurrency formalism. The programming model should provide useful modeling properties that the designer cares about. It should be easy to reason about those properties. Furthermore, it should yield static analyzability whenever possible so that some form of correctness can be guaranteed during design time.

Ptolemy II is a platform to try out programming models with different concurrency formalisms. Unlike other design frameworks, the Ptolemy project is unique in the breadth of exploration of semantic alternatives. A rich variety of concurrency formalisms span the whole spectrum from most expressive to most analyzable. In particular, I will focus on one particular model of computation called heterochronous dataflow (HDF) which strikes a nice balance between expressiveness and analyzability.

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

Since the beginning the Ptolemy II design environment emphasizes flexibility over efficiency. E.g., in order to support run-time reconfiguration of Ptolemy models, many operations have extra levels of indirection. The overhead from this indirection

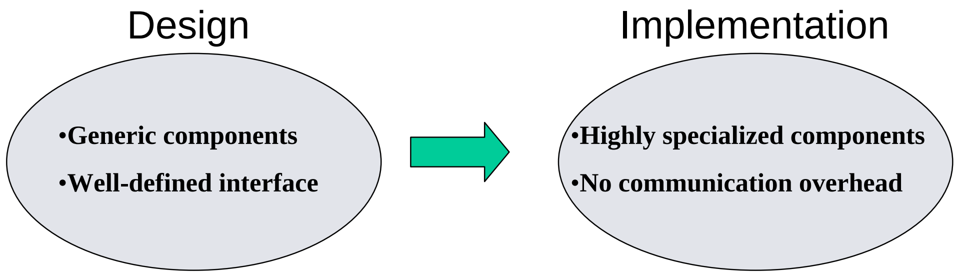


Figure 1.3: From design to implementation.

is incurred in all models, even if a particular model does not use reconfiguration [Neuendorffer2004]. Ptolemy II introduces the notion of domain-polymorphic and data-polymorphic actors so that they can be used for different models of computation and with different data types. Like any generic components, actors designed this way trade efficiency for flexibility. All these features contribute to the ease of use of Ptolemy II as a rapid prototyping environment. The question is how to get back the efficiency so that we can turn a design prototype into an implementation (see Fig. 1.3), which is the theme of the second part of this thesis, corresponding to the bottom arrow in Fig. 1.2.

To regain the efficiency for the implementation, the users could write big monolithic components to reduce inter-component communication, and write highly specialized components rather than general ones. However, manually implementing these solutions is not an option. Partial evaluation [Jones1993] provides a mechanism to automate the whole process. The basic idea is that given a program and part of this program's input data, a partial evaluator can execute the given program as far as possible producing a residual program that will perform the rest of computation when the rest of the input data is supplied. It usually involves a binding-time analysis phase to determine the static parts and the dynamic parts of a program, followed by an evaluation phase. The derived program is usually much more efficient by removing computation overhead resulting from the static parts.

Partial evaluation has been in use for many years and applied in a variety of programming languages including functional languages [Gomard1991], logic languages [Lloyd1991], imperative languages like C [Anderson1992] and object-oriented languages [Schultz2001]. Its use in embedded software has been more recent, as seen in Click component framework [Kohler2002], Koala component model [Ommerling2000], etc. Compared with previous examples, Ptolemy II does not focus on specific applications. Its emphasis is on choosing appropriate concurrent MoC's for embedded system design and generating efficient implementation for the chosen MoC's.

In my research, partial evaluation is used as a code generation technique, which is really a compilation technique for transforming an actor-oriented model into the target code while preserving the model's semantics. However, compared with traditional compiler optimization, partial evaluation for embedded software works at the component level and heavily leverages domain-specific knowledge. Through model analysis—the counterpart of binding-time analysis in traditional use of partial evaluation for general purpose software—the tool can discover the static parts in the model including data types, buffer sizes, parameter values, model structure and model execution schedules, etc. and then evaluate all the known information to reach a very efficient implementation. The end result is that the benefit offered by the high level abstraction comes with (almost) no performance penalty.

Chapter 2

Model-based Design

In [Poore2004] Poore compared the fields of circuit engineering and software engineering.[1] He noted that the field of circuit engineering fully exploits the Boolean algebra associated with circuits; in fact, the entire process of circuit design is a mathematical activity, and the hardware mirrors the algebraic operations AND, OR, and NOT. A designer starts the design by formulating the new function at the highest level of abstraction and proceeds by refining and assembling pre-defined components. The final generated implementation is correct by design, equivalent to the original rigorous mathematical formulation. In another words, it works correctly the first time it is used (or tested) because it is designed and built right. Only rank amateurs in their home radio shacks would construct devices without first sketching diagrams or scribbling functions. Therefore they have to resort to trial and error to make sure

[1]The other discipline in the paper is genetic engineering. However, discussing the difference between the two disciplines, i.e., circuit engineering and software engineering, serves the purpose for this thesis.

constructed devices work. Unfortunately the way software is usually written is a bit like how amateurs construct their devices. Instead of generating correct-by-design implementation from a rigorous mathematical formulation, the software is constructed from bottom up and extensive tests are needed to make sure components in a software system work together. The situation is even worse for multi-threaded software, where some small change often requires extensive re-testing.

Like circuit engineering, model-based design takes a top-down approach and is based on a strict mathematical formulation. Taking Simulink [Simulink] for an example, its design framework incorporates the four phases of a design cycle: model a plant; analyze and synthesize a controller for the plant; simulate and verify the correctness of the controller; deploy the controller. This model-based design framework is significantly different from the traditional design methodology. Instead of starting design by writing software code, designers can now start building models by defining advanced functional characteristics based on underlying mathematical formulation of the framework. These built models along with some simulation and verification tools can lead to rapid prototyping, software testing, hardware-in-the-loop simulation and verification.

However, there are differences. Circuit engineering is based on one mathematical formulation, the Boolean algebra. Embedded software, on the other hand, is often domain-specific and highly specialized. An appropriately chosen formalism can help a designer focus on the functional aspect of a design instead of accidental issues imposed by the formalism. For example, a signal processing application is naturally described by a dataflow formalism. On the other hand, FSM's are best used to express control logic. A mismatch between the application and the formalism will lead to an awkward design process.

Thus choosing a good design framework with an appropriate formalism is the prerequisite for doing a good design. In [Lee2005] Lee outlines the criteria for a good framework: it expresses the important properties of a design; it supports modularity and abstraction which scales; it complies (or code generates) to cost-effective solutions; and designs built using it are understandable and analyzable.

2.1 Actor-Oriented Design and Concurrent Models of Computation

A number of design frameworks have emerged over the years that offer different concurrency models for the applications they support. For example, StreamIt [Thies2002] has a dataflow formalism nicely matched to streaming media applications. Simulink [Simulink] has roots in control system modeling, and time is part of its formal semantics. All these frameworks have formal concurrent models of computation (MoC's) that match their application spaces. They often use block diagram based design environments (e.g., see Fig. 2.1) and the design usually starts with assembling preexisting components in the library. This kind of model-based design

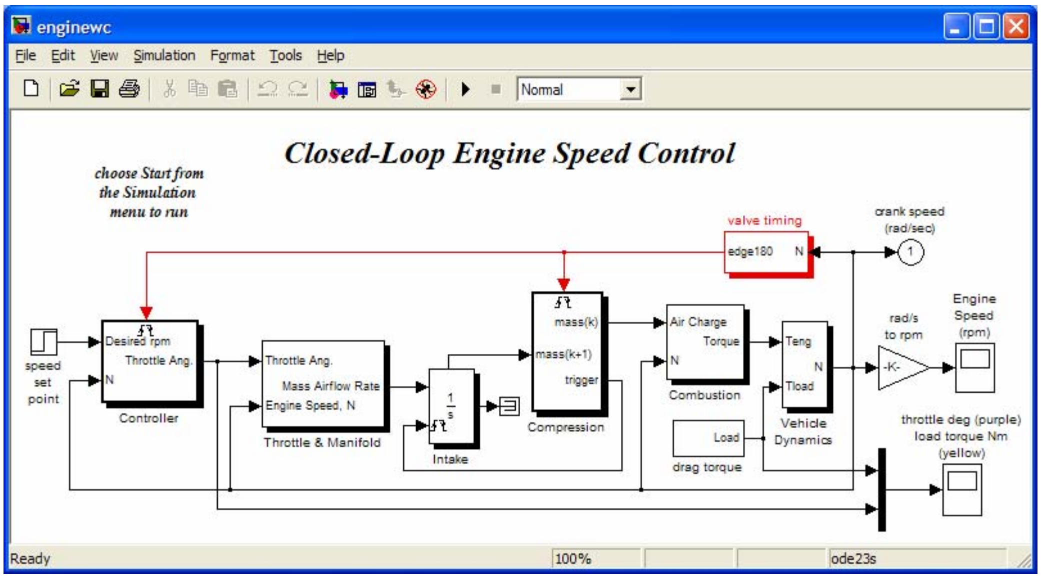


Figure 2.1: A Simulink block diagram modeling a plant and a controller.

style has also been called domain specific design, or component based design, each emphasizing different aspects of the design.

Many of these design frameworks employ an actor-oriented approach, where actor is an encapsulation of parameterized actions performed on input data to produce output data. Input and output data are communicated through well-defined ports. Ports and parameters form the interface of an actor. Fig. 2.2 gives a comparison between object-oriented design and actor-oriented design [Lee2005]. In object-oriented design, the interactions between objects are method-call based and what flows through an object is sequential control. In actor-oriented design, the interactions between actors are message-passing based and what flows through an actor is streams of data. Although both object-oriented design and actor-oriented design emphasize data encapsulation

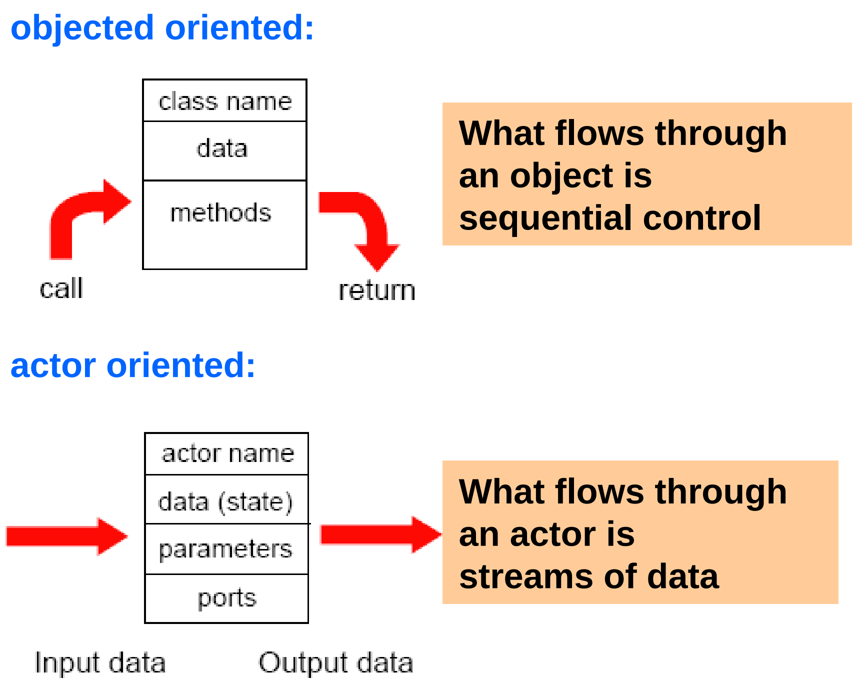


Figure 2.2: object-oriented design vs. actor-oriented design [Lee2005].

and make the data of one object/actor inaccessible to other objects/actors, they are fundamentally different. In object-oriented design, the calling object still manipulates the data of the called object through method calls and object activation is sequentialized. In actor-oriented design, the upstream actor sends a message to the downstream actor and it's up to the downstream actor to decide how to consume the passed messages. This emphasis of data flow over control flow leads to conceptually concurrent execution of actors.

Under the umbrella of actor-oriented design, there is a rich variety of models of computation that deal with concurrency and time in different ways. Each gives an interaction mechanism for components. In the introduction chapter of [Brooks2007-1], it gives a nice summary of the way various models of computations are used. It states that the utility of a model of computation stems from the modeling properties that apply to all similar models. For many models of computation these properties are derived through formal mathematics. Depending on the model of computation, the model may be determinate [Kahn1974], statically schedulable [Lee1987-2], or time safe [Henzinger2002]. Because of its modeling properties, a model of computation represents a style of modeling that is useful in any circumstance where those properties are desirable. In other words, models of computation form design patterns of component interaction, in the same sense that design patterns are used in object-oriented languages [Gamma1994].

For a particular application, an appropriate model of computation does not impose unnecessary constraints, and at the same time is constrained enough to result in useful derived properties. For example, by restricting the design space to synchronous designs, Scenic [Liao1997] enables cycle-driven simulation [Hansen1988], which greatly improves execution efficiency over more general discrete-event models of computation (such as that found in VHDL). However, for applications with multirate behavior, synchronous design can be constraining. In such cases, a less constrained model of computation, such as synchronous dataflow [Lee1987-2] or Kahn process networks [Kahn1974] may be more appropriate. One drawback of this relaxation of synchronous design constraints is that buffering becomes more difficult to analyze. On the other

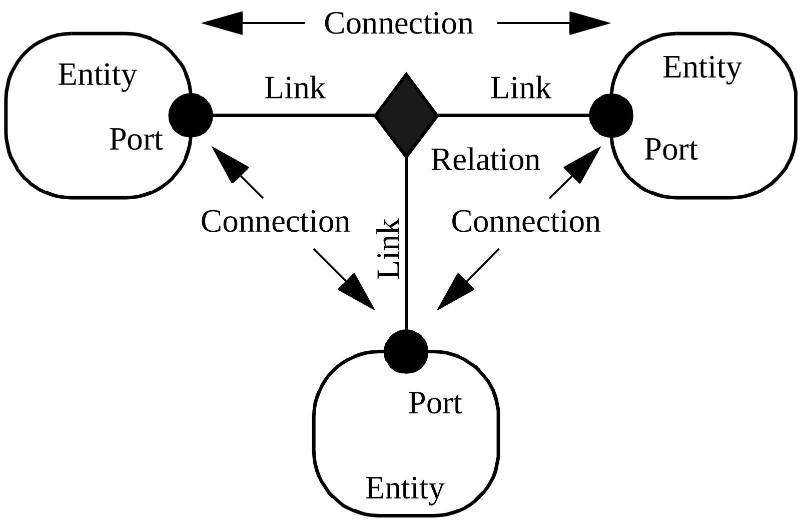


Figure 2.3: Abstract syntax of Ptolemy II models.

hand, techniques exist for synchronous dataflow that allow co-optimization of memory usage and execution latency [Teich1999] that would otherwise be difficult to apply to a multirate system. Selecting an appropriate model of computation for a particular application is often difficult, but this is a problem we should embrace instead of avoiding.

2.2 Ptolemy II: a Software Environment for Experimenting with MoC's

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. Unlike most system design environments which have one or a limited number of built-in models of computation (e.g., Simulink is originally based purely on continuous time semantics[2] and later added Stateflow, which has finite state machine semantics, and SimEvents, which has discrete event semantics [Simulink]; StreamIt is based purely on dataflow semantics [Thies2002]), the Ptolemy kernel has no built-in semantics. Instead the kernel package defines a small set of Java classes that implement a data structure supporting a general form of uninterpreted clustered graphs (see Fig. 2.3 taken from [Brooks2007-2]), which provide an abstract syntax that is not concerned with the meaning of the interconnections of components, nor even what a component is.

The semantics is introduced in the actor package which provides basic support for executable entities. However, it makes a minimal commitment to the semantics of these entities by avoiding specifying the order in which actors execute and the communication mechanism between actors. These properties are defined in each MoC where the director controls the execution of actors and the receiver controls the communication between actors (see Fig. 2.4)

In Ptolemy II, an MoC is called a domain.[3] The domains currently implemented include: Component Interaction (CI), Communicating Sequential Process (CSP),

[2]Simulink's discrete time signal is actually piecewise constant continuous time signal.

[3]Note the different usage of the term *domain*. Here domain is PtII-specific, referring to an MoC. In domain-specific modeling, domain refers to a specific application area. They can be loosely related though, considering that different domain-specific modeling environment may need different MoC's (PtII domains).

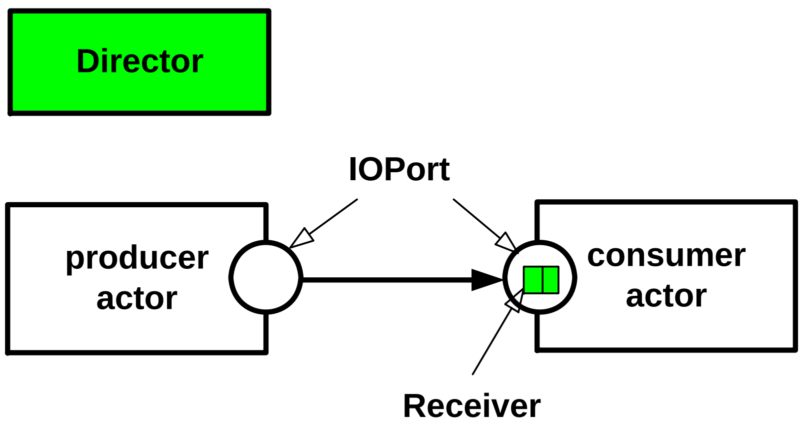
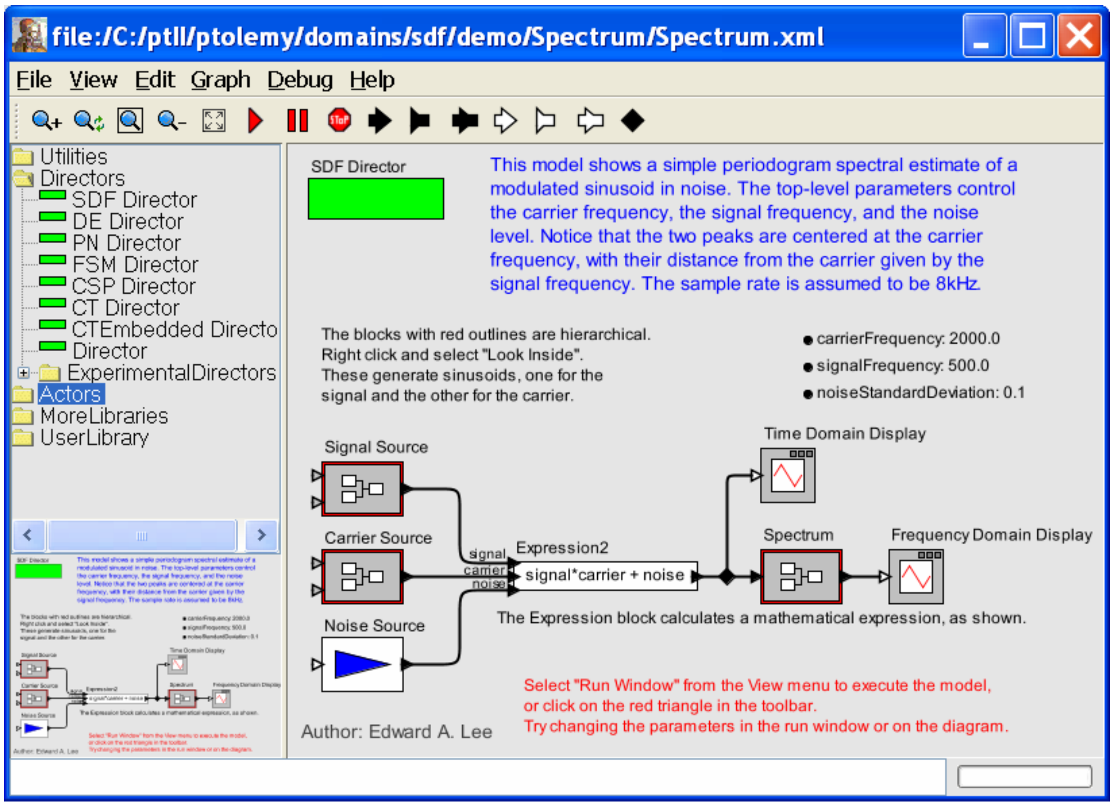


Figure 2.4: The semantics of a Ptolemy II domain is defined by director and receiver.

Continuous Time (CT), Distributed Discrete Event (DDE), Dynamic Dataflow (DDF), Discrete Event (DE), Discrete Time (DT), Finite State Machines (FSM), Giotto, Graphics (GR), Heterochronous Dataflow (HDF), Process Networks (PN), Parameterized Synchronous Dataflow (PSDF), Synchronous Dataflow (SDF), Synchronous Reactive (SR), and Timed Multitasking (TM).

An essential difference between concurrent models of computation is their modeling of time [Lee2006]. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line (such as CT, DE). Others are more abstract and take time to be discrete (such as DT, SR). Others are still more abstract and take time to be merely a constraint imposed by causality (such as PN, CSP, SDF, DDF). The last interpretation results in time that is partially ordered, which provides a mathematical



This model shows a simple periodogram spectral estimate of a modulated sinusoid in noise. The top-level parameters control the carrier frequency, the signal frequency, and the noise level. Notice that the two peaks are centered at the carrier frequency, with their distance from the carrier given by the signal frequency. The sample rate is assumed to be 8kHz.

The blocks with red outlines are hierarchical. Right click and select "Look Inside". These generate sinusoids, one for the signal and the other for the carrier.

- carrierFrequency: 2000.0
- signalFrequency: 500.0
- noiseStandardDeviation: 0.1

The Expression block calculates a mathematical expression, as shown.

Select "Run Window" from the View menu to execute the model, or click on the red triangle in the toolbar. Try changing the parameters in the run window or on the diagram.

Figure 2.5: Vergil — a graphical user interface for Ptolemy II.

framework for formally analyzing and comparing models of computation [Lee1998].

There is a large actor library in Ptolemy II. Some actors are written to be domain-polymorphic, i.e., they can operate in any of a number of domains. They are the result of the principle of separation of function and communication. Some actors can only be used in specific domains because they deal with functionalities specific to those domains. Fig. 2.5 shows a graphical user interface in Ptolemy II called Vergil where designers compose models by assembling pre-built actors and choosing appropriate directors.

There are three big volumes of design documents [Brooks2007-1][Brooks2007-2] [Brooks2007-3] for Ptolemy II. Volume I gives an overview of the Ptolemy II design environment and serves as a user's manual. Volume II covers details of the software architecture and serves as a developer's manual for those who want to extend Ptolemy II design environment. Volume II covers various domains implemented in Ptolemy II. These documents are usually updated with each software release.

Chapter 3

HDF: a Decidable Dataflow Formalism

3.1 Motivation for Dataflow Computing

The Dataflow model of computation is most popularly used in digital signal processing (DSP). Dataflow programs for signal processing are often described as directed graphs where each node represents a function and each arc represents a signal path. Compared with imperative programs, which do not often exhibit the concurrency available in the algorithm, dataflow programs automatically break signal processing tasks into subtasks and thus expose the inherent concurrency as a natural consequence of programming methodology. In a dataflow graph, each function node executes concurrently conceptually with the only constraint being data availability. Therefore it greatly facilitates efficient use of concurrent resources in the implementation phase.

Dataflow programming has come a long way since its use in the signal processing community. Various programming environments based on this model of computation have been developed over the years. Various specialized dataflow models of computation have been invented and studied. Various scheduling algorithms based on different criteria pertinent to specific applications have been proposed. It is still a fruitful area of active research due to its expressive way to represent concurrency for embedded system design.

Over the years, a number of dataflow models of computation (MoC) have been proposed and studied. In the synchronous dataflow (SDF) MoC studied by Lee and Messerschmitt [Lee1987], each dataflow actor consumes and produces fixed number of tokens on each port in each firing. The consequence is that the execution order of actors can be statically determined prior to execution. This results in execution with minimal overhead, as well as bounded memory usage and a guarantee that deadlock will never occur. The cyclo-static dataflow MoC [Engels1994] extends SDF by allowing each actor's consumption and production rates to vary in a cyclic but predetermined pattern. It does not add more expressiveness to SDF, but turns out to be more convenient to use in some scenarios. The heterochronous dataflow MoC (HDF) proposed by Girault, Lee and Lee [Girault1999] extends SDF by using so-called modal model to compose FSM with SDF. It allows actors to change their rate signatures between global iterations of a model. In case of a finite number of rate signatures, properties like consistency and deadlock are still decidable. Bhattacharya and Bhattacharya proposed a parameterized synchronous dataflow (PSDF) MoC [Bhattacharyaand2000], which is useful for modeling dataflow systems with reconfiguration. In this domain, symbolic analysis of the model is used to generate a quasi-static schedule that statically determines an execution order of actors, but dynamically determines the number of times each actor fires. Buck proposed a Boolean dataflow (BDF) MoC [Buck1993] which allows the use of some dynamic actors such as BooleanSelect and BooleanSwitch. He extends the static analysis techniques used in SDF and in some situations a quasi-static schedule can be pre-computed. But fundamentally since BDF is Turing-complete, it does not guarantee that the scheduling algorithm will always succeed. If it fails, a dynamic dataflow (DDF) MoC [Parks1995] should be used to execute the model. DDF uses only runtime analysis and thus makes no attempt to statically answer questions about deadlock and boundedness. Since this section only serves as a brief review, it suffices to say that the list here does not include every variant of dataflow ever conceived.

3.2 HDF

Almost all general purpose programming languages in widespread use, such as Java and C, are Turing-complete. One well known result in computability theory is that the halting problem is undecidable over Turing machines [Sipser1996]. Furthermore, Rice's theorem states that any nontrivial property about the language recognized by a Turing machine is undecidable. Fundamentally, that is the reason why debugging is always a difficult task since there are no general algorithms for doing that. In practice, programmers mostly rely on testing to uncover bugs in the software and achieve some degree of quality assurance. However, software testing remains a dark art [Myers2004]. Myers et al stated in their book *The Art of Software Testing* [Myers2004] that "at the time this book was first published, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed; today, a quarter of the century later, the same is still true".

In fact, things could get much worse when testing the embedded software, which has seen a drastic increase in complexity in recent years. For one thing, timing properties are at least as important as the function (as implemented on a Turing machine) in determining the correctness of an embedded system. However, the timing properties of a system are notoriously difficult to test, often with a small deviation from the expected operating condition resulting in missing deadlines and catastrophic failures. Furthermore, a recent trend of embedded systems is to move from closed "boxes" that do not expose the computing capability to the outside, to deeply embedded network systems. Such networking poses considerable technical challenges in a networked environment, and it becomes impossible to test the software under all possible conditions [Lee2008].

On the other hand, reliability standards for embedded software are much higher than general-purpose software. In desktop computing, if a program is deadlocked, or runs out of memory, restarting the program has become an acceptable way of “fixing” the problem, no matter how inconvenient it is. In embedded computing, people have much higher expectations. They expect programs simply to work and expect them to run continuously for years without problems. With software embedded into many devices today—including high confidence medical devices and systems, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control, distributed robotics, defense systems, manufacturing, and smart structures [Lee2008]—software failure causes more than inconvenience. It could cause human fatalities, or affect a large population, or result in huge financial loss. Even for the embedded systems as simple as those as in household appliances, such as microwave ovens, washing machines and dishwashers, people expect them to work each time. (It’s interesting to see that people set the bar much higher for those that are not first-and-foremost computers than the computers sitting on the desktop.) So is there an alternative way to design such system to meet high reliability standards?

One solution to solve the difficulty of comprehensive testing is to be correct by design. Unlike traditional general purpose programming languages, which strive to provide all-inclusive one-stop solution for all applications, embedded system design is domain-specific and highly specialized. Designers can choose a formalism appropriate for the specific applications to more effectively express the design (this is the essence of the previous chapter on model-based design). Moreover designers can carefully choose a decidable formalism, which makes it possible to check certain key properties during compilation time. Some important properties for embedded systems, such as deadlocks, a bound on memory consumption and a bound on execution time, can be statically analyzed if these properties are indeed decidable from the formalism.

For example, synchronous languages, including Esterel [Boussinot1991], Lustre [Halbwachs1991] and Signal [Le Guernic1991], have been established as a technology of choice for modeling, specifying, validating, and implementing real-time embedded applications [Halbwachs1993][Benveniste1991][Benveniste2003]. They have best found their way in the design of embedded control systems. Since they are based on a decidable formalism, designers can formally reason about the key properties of the system during the design stage. Taking Esteral as an example, he/she could determine whether a program is deadlock-free by doing so-called causality analysis. Furthermore, an Esterel program is equivalent to a finite state machine and therefore verification techniques can be used to prove the correctness of the program.

The SDF formalism is another decidable MoC amenable to static analysis. SDF is a multi-rate dataflow language. Each actor in an SDF diagram has a certain number of input and output ports, each labeled with the number of data tokens the actor consumes (hence called consumption rate) or produces (hence called production rate) on the port each time the actor fires. The set of production rate of all output ports and consumption rate of all input ports for an actor is called the rate signature of

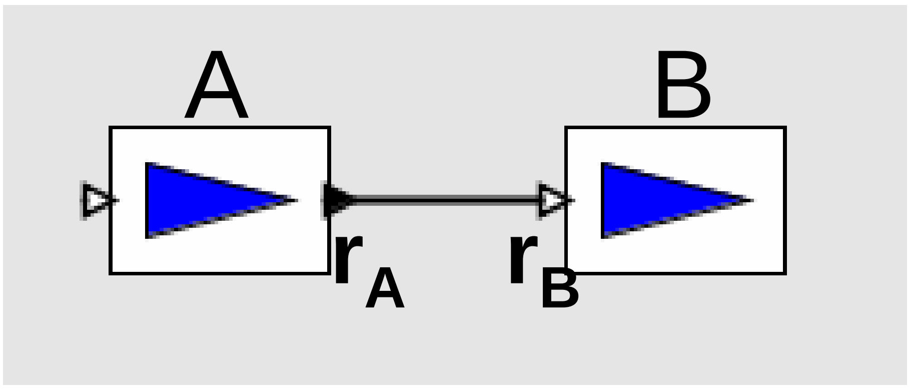


Figure 3.1: A simple SDF block diagram.

this actor. The decidability of the SDF formalism lies in the *fixed* rate signature of each actor.

In the simple SDF block diagram shown in Fig. 3.1, actor A has a production rate of 2 for its output port and actor B has a consumption of 3 for its input port:

$$r_A = 2, \; r_B = 3.$$

Ports are connected by first-in first-out buffers. By solving the following balance equation, which actually is a constraint imposed by the relative consumption and production rates on the buffer:[1]

$$f_A r_A = f_B r_B,$$

we can determine the number of firings for actor A (f_A) and actor B (f_B) in one periodic firing schedule:

[1]For a general SDF diagram, one needs to solve multiple balance equations since each connection imposes a constraint and thus results in one equation.

$$f_A = 3,\ f_B = 2.$$

The next step is to construct the actual executable schedule that follows data precedence given the solution to the balance equations. In this particularly simple example, there are two feasible schedules: AAABB or AABAB. Each schedule requires different code size and buffer size. For a complicated SDF graph, one could imagine there is great scheduling flexibility and much work has been done on scheduling it efficiently [Bhattacharyya1996]. By repeating the schedule (we call one firing of the schedule one complete iteration), the system will never deadlock and the consumed memory will always be bounded.

On the other hand, if there is no non-zero solution to the balance equations, or if there is non-zero solution to the balance equations but no correspondent schedule to realize it, then the system either cannot be executed in bounded memory due to rate inconsistency or the system will deadlock due to the lack of initial data tokens. Thus designs can be improved upon and errors can be corrected while the system is still in the initial design stage so that there is no surprise at run time.

The important lesson learned here is we limit expressiveness in exchange for considerable advantages such as compile-time predictability. The SDF formalism provides such an advantage when the system under design can be expressed as such—no dynamic behavior, no reconfiguration. Otherwise more expressive formalisms are needed. However, going to the other side of the spectrum with a Turing-complete language will lose all the advantage of decidability. The key is to find a sweet spot where we could accommodate both in some form.

Finite state machines (FSM's) have been the subject of a long history of research work and are often used to describe and analyze intricate control sequences. Due to their finite nature, a designer can enumerate the set of reachable states to ascertain that a particularly dangerous state cannot be reached. In [Girault1999] Girault, Lee and Lee propose combining FSM's with multiple concurrency models to form so-called *charts. Other approaches to combine FSM's with concurrent models of computation include the original statecharts [Harel1987], the Argos language [Maraninchi1991] combining FSM's with a Synchronous/Reactive (SR) concurrency model, the Specification and Description language (SDL) [Belina1991] combining FSM's with Process Networks (PN), the codesign finite state machine (CFSM) [Chiodo1994] combining FSM's with a discrete-event (DE) concurrency model, the Hybrid systems [Alur1995] combining FSM's with concurrent continuous-time models etc. These approaches tightly intertwine the concurrency model with the automata semantics, and most of them have limited compositionality in that they permit automata only in the leaf cells or at the top of the hierarchy. The innovation in the *charts approach is to decouple the concurrency model from the hierarchical FSM semantics so that designers can choose an appropriate concurrency model to match the problem at hand. Moreover, the hierarchy in *charts can be arbitrarily deep, and concurrency models and FSM's can be placed anywhere within it. An FSM can be nested within a module in a con-

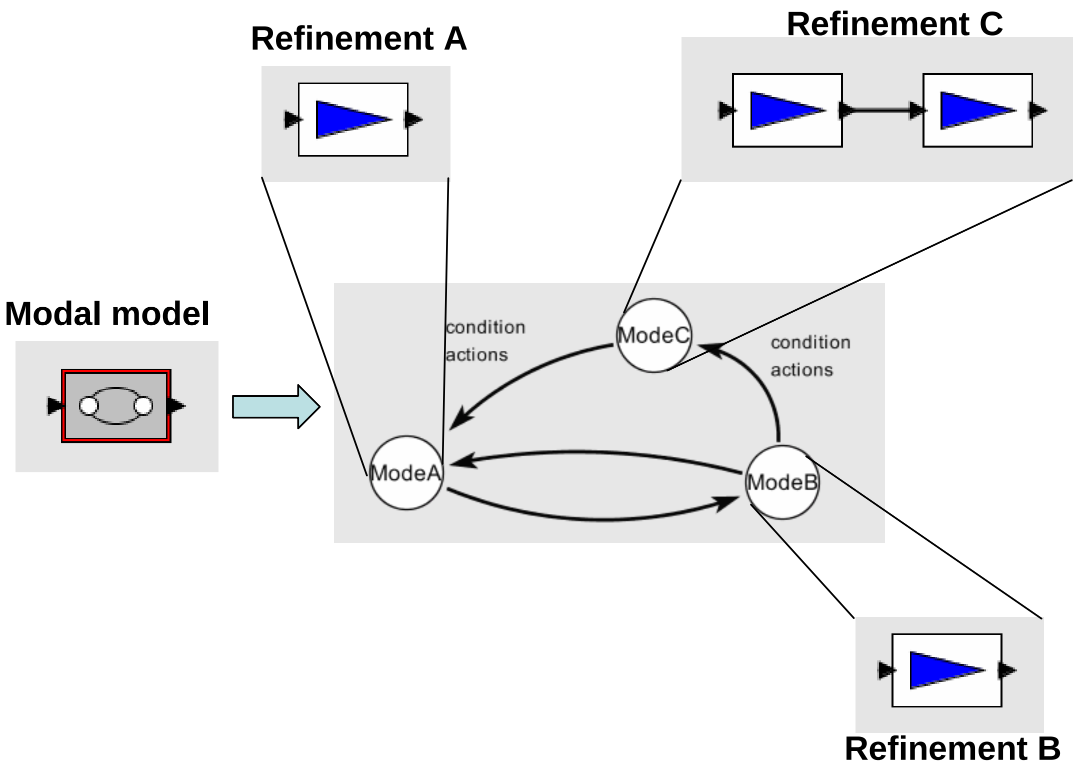
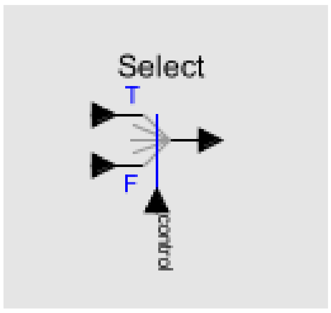


Figure 3.2: A modal model showing refinement for each state.

currency model, with the interpretation that the FSM describes the behavior of the module. Conversely, a subsystem in some concurrency model can be nested within a state of an FSM, with the meaning that the subsystem is active if and only if the FSM is in that state. The latter is particularly well suited to describing modal systems, where modes of operation are modeled as states of an FSM.

Fig. 3.2 shows a modal model actor with three states where each state contains a refinement that describes the behavior of the modal model actor when it is in that state. With each state refined by an SDF sub-model, we have a convenient, compact



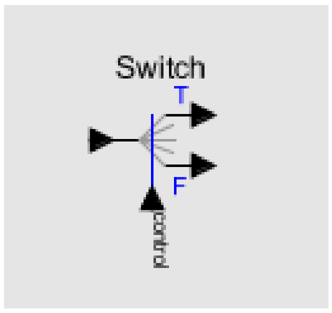


Figure 3.3: Select and Switch actor.

and understandable way to express modal behavior by combining FSM with SDF. The next question is whether we lose compile-time predictability if we express system design using this formalism.

To answer this question, let's analyze two actors in Fig. 3.3. The Select actor first reads a boolean token from its control port. Depending on the value of the boolean token, it either reads a data token from its T port (if the boolean token is true) or from its F port (if the boolean token is false) and then sends the data token to its output port. The Switch actor has a similar behavior. It first reads a boolean token from its control port. Depending on the value of the boolean token, it reads a data token from the other input port and sends the data token either to its T port (if the boolean token is true) or to its F port (if the boolean toke is false).

We can model the behavior of Select and Switch actors with modal models. Fig. 3.4

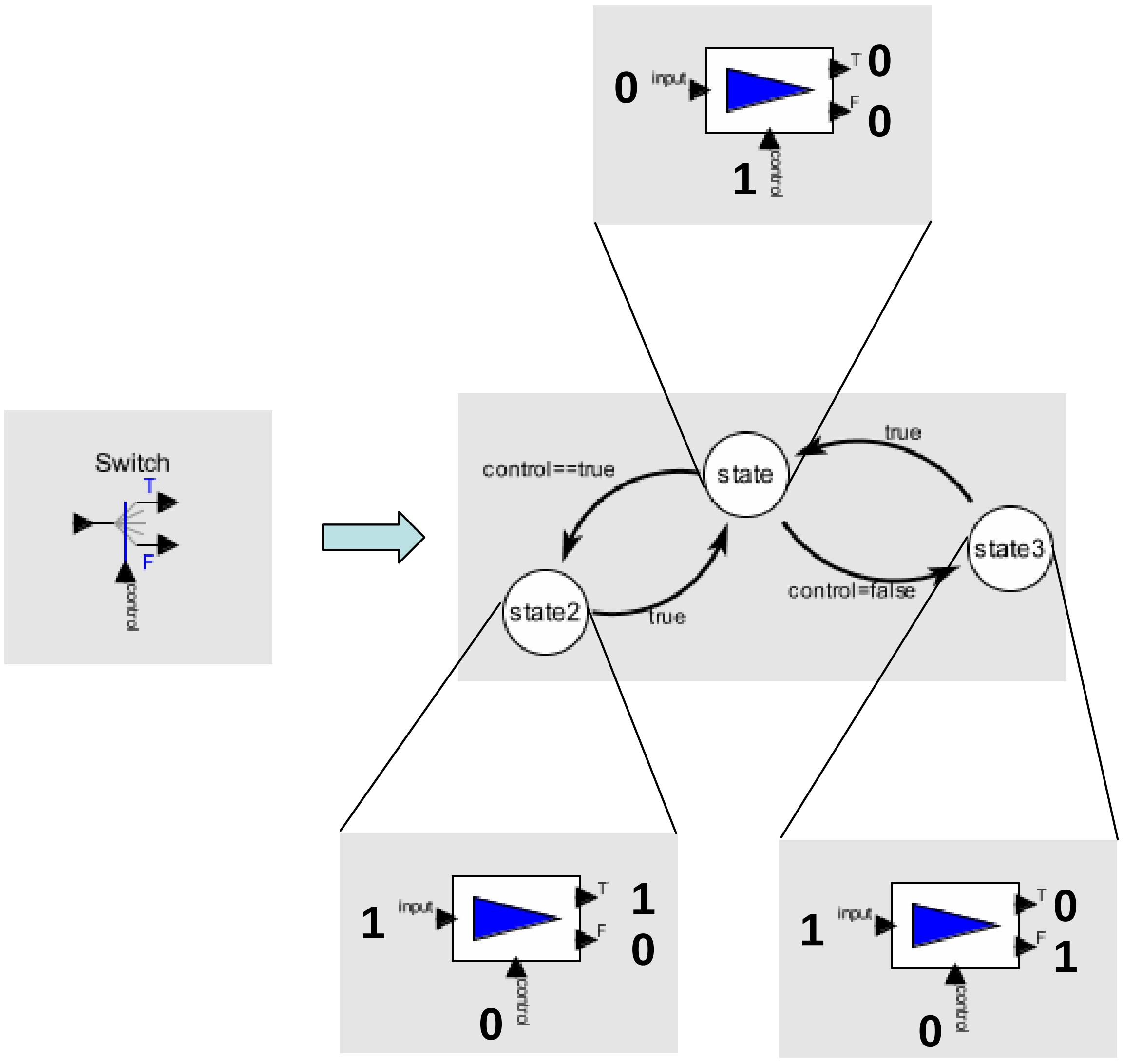


Figure 3.4: Switch actor as a modal model.

provides an equivalent implementation for the Switch actor. In the refinement for the initial “state”, the control port has rate 1 and all other ports have rate 0, so only one token is read from the control port. The next step is to evaluate all guard expressions for the state transitions from the initial “state” and take the transition to either “state2” or “state3”. According to their rate signatures, the refinement for “state2” reads one token from the input port and sends it to T port; the refinement for “state3” reads one token from the input port and sends it to F port. Eventually the state machine takes the transition back to the initial “state” since the guard expression is always true for the transition from “state2” or “state3”. And so begins the next cycle. The Select actor can be modeled similarly.

To answer the compile-time predictability question, we need the following theorem:

Theorem 1 *[Buck1993] BDF (boolean data flow) models consisting the Select and Switch actors, together with actors for performing addition, subtraction, and comparison on the integers, plus a source actor that produces constant stream of integer-valued tokens and a fork actor, are Turing equivalent.*

Since we have shown that with the composition of FSM and SDF we can implement the Select and Switch actors, it naturally follows that:

Corollary 1 *The composition of FSM and SDF creates a Turing-complete language.*

Since for a Turing-complete language, deadlock and bounded memory consumption are undecidable, we also have:

Corollary 2 *For designs that are expressed with FSM + SDF formalism, deadlock and bounded memory consumption are undecidable.*

Then the next natural question is what constraint it takes for the formalism to be decidable. For a model that is a composition of FSM and SDF, when each FSM takes a transition to a certain state, we can replace the FSM with the refinement for that state. So if we take a snapshot after any state transition, the model would look like an SDF graph. Since it takes a complete iteration of an SDF graph to return buffers to their original size, the key to retain decidability is to only allow state transitions to occur after one complete iteration. With this constraint, the resulting MoC is called heterochronous data flow (HDF), first proposed in [Girault1999].

Fig. 3.5 show a typical HDF model. At each level of the hierarchy, the name inside the green box represents the director (Ptolemy II terminology) implementing the MoC at that level of the hierarchy. At the top level is an HDF model. Two actors at that level are refined by FSM's. For the FSM on the right hand side, the two states are refined by SDF sub-models. For the FSM on the left hand side, one state is refined by SDF sub-model and the other state is refined by HDF sub-model. The actor in the HDF sub-model is further refined an FSM and so it goes on. The execution of the HDF model goes as the following: after the top level model finishes one complete iteration, each active[2] FSM evaluates the guard expressions for the arcs

[2]An FSM is not active if it descends from the state ϕ of another FSM' and ϕ is not the current state of FSM'. For example, in Fig. 3.5 the two FSM's right below the top level are always active. The third FSM below can be active or not depending on the current state of the FSM above it.

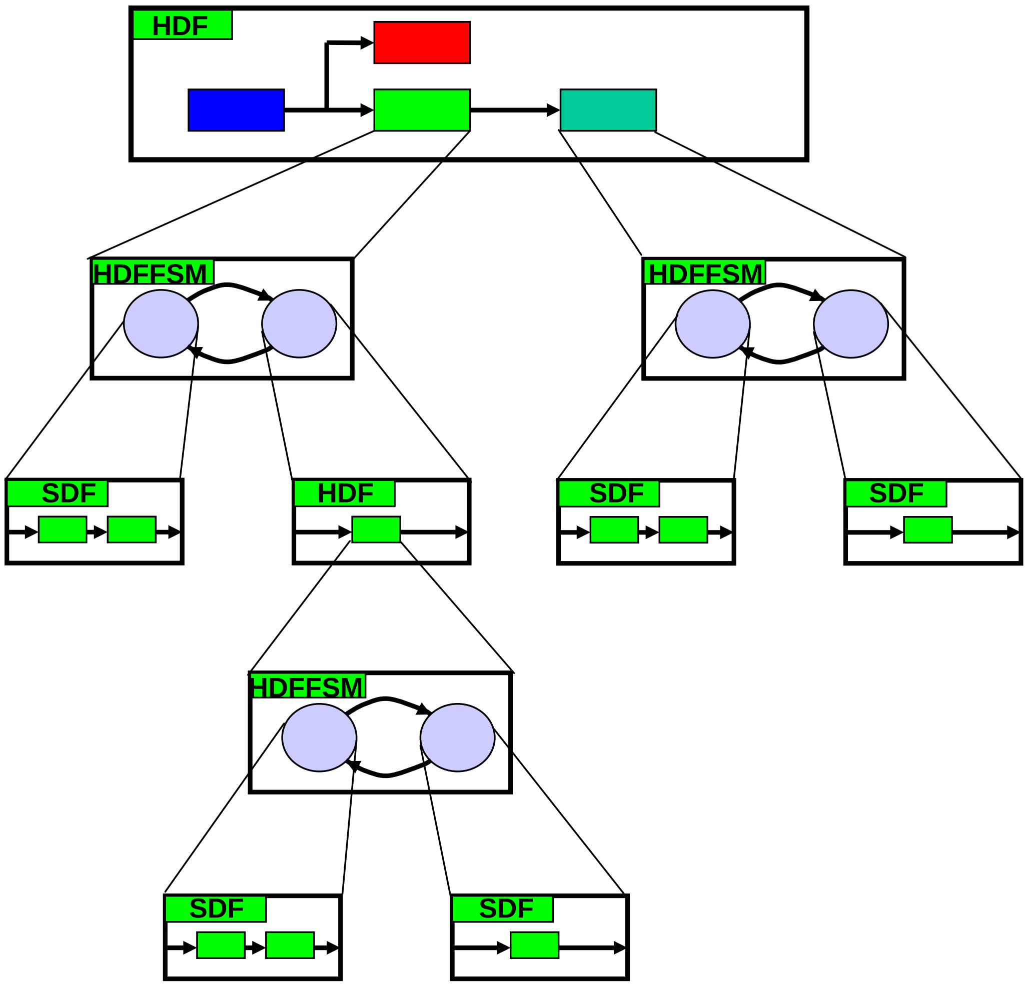


Figure 3.5: A hierarchical HDF model.

emanating from the current state and makes the transition if the corresponding guard expression evaluates to true. The rate signatures of some actors may change after making the state transitions and a new SDF schedule is generated or retrieved (if pre-calculated). The next iteration continues with the new schedule and the cycle repeats. Due to the finite nature of FSM's, we can pre-analyze all possible SDF schedules and determine deadlock and memory-consumption for each schedule. Therefore we have the following theorem:

Theorem 2 *HDF is not Turing-complete.*

Proof In HDF, deadlock is decidable, which means the halting problem for HDF is decidable. Since for any Turing-complete formalism the halting problem is not decidable, HDF cannot be Turing-complete.

3.3 Configuration Analysis

Fig. 3.6 shows an execution trace for an HDF model. It alternates between doing one complete iteration and making state transtions. Between state transitions, it can be reduced to an SDF model. If the SDF model has a non-zero solution to its balance equations and there exists corresponding schedule, we say the model is in a well-defined configuration.

Since an HDF model can have an arbitrary number of levels of hierarchy, there must be a systematic way to find out the number of configurations and derive the

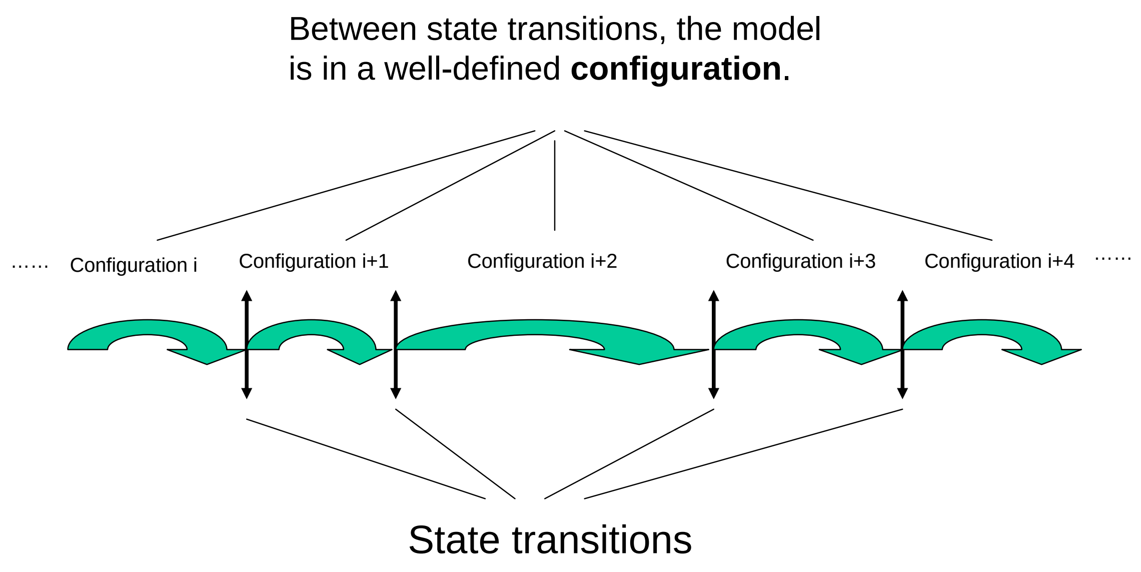


Figure 3.6: An execution trace for HDF model.

schedule for each configuration. The approach taken here uses the following definitions.

For each opaque actor (i.e., atomic actor or opaque composite actor[3]), let N be the number of configurations of this actor. For each configuration of the actor, it has a corresponding local SDF schedule.

Let r_{ij} be the rate of the jth port of this actor in the ith configuration, where $i = 0, \ldots, N-1$, $j = 0, \ldots, P-1$, and P is the number of ports of this actor.

Let $R_i = \{r_{ij} | j = 0, \ldots, P-1\}$ be the rate signature of this actor in the ith configuration.

During code generation, N and R_i, $i = 0, \ldots N-1$, for each opaque actor are

[3]An opaque composite actor is a composite actor containing a local director. Thus the MoC used inside an opaque composite actor can be different from the MoC's used higher up in the hierarchy. It is the mechanism for building models with hierarchically coupling heterogeneous MoC's in Ptolemy II.

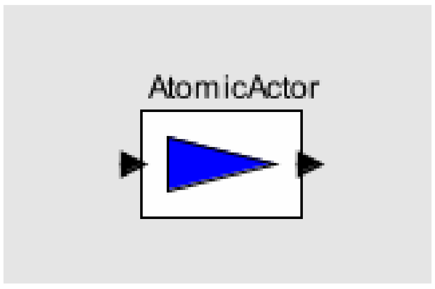


Figure 3.7: Configuration analysis for an atomic actor.

derived in a recursive bottom-up fashion:

1) *Atomic actor* (see Fig. 3.7):

$$\begin{aligned} N &= 1 \\ R_0 &= \text{the rate signature of the atomic actor} \end{aligned}$$

Comments: an atomic actor has a degenerate form of local SDF schedule: fire itself once.

2) *Composite actor with a local SDFDirector* (see Fig. 3.8):

$$\begin{aligned} N &= 1 \\ R_0 &= \text{the rate signature of the composite actor} \\ &\quad\ \text{inferred from the local SDF schedule} \end{aligned}$$

Comments: each contained actor can only have one rate signature (i.e., one configuration). Otherwise the local director should be a HDFDirector. In Fig. 3.8, after

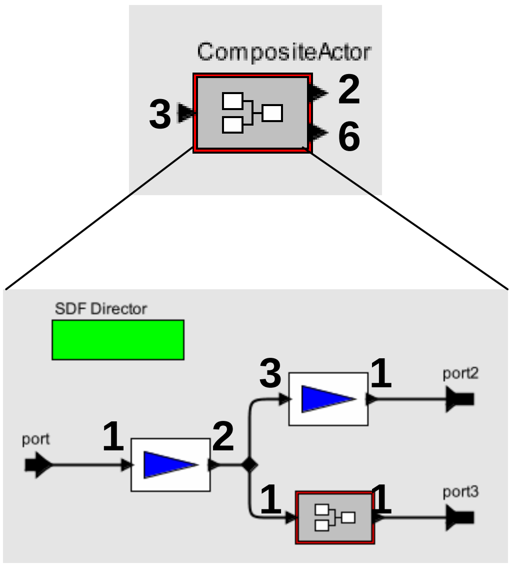


Figure 3.8: Configuration analysis for a composite actor with a local SDFDirector.

determining the local schedule, we can determine the number of data tokens consumed and produced by the composite actor to complete a local iteration, and that is the rate signature for the composite actor.

3) *Modal model with a local HDFFSMDirector* (see Fig. 3.9):

$$N = \sum_{j=0}^{M-1} N_j$$

where N_j is the number of configurations of the refinement for the jth state, and M is the number of states.

For $i = 0, \ldots, N-1$,

$$R_i = \text{the rate signature of the } k\text{th refinement in its } q\text{th configuration}$$

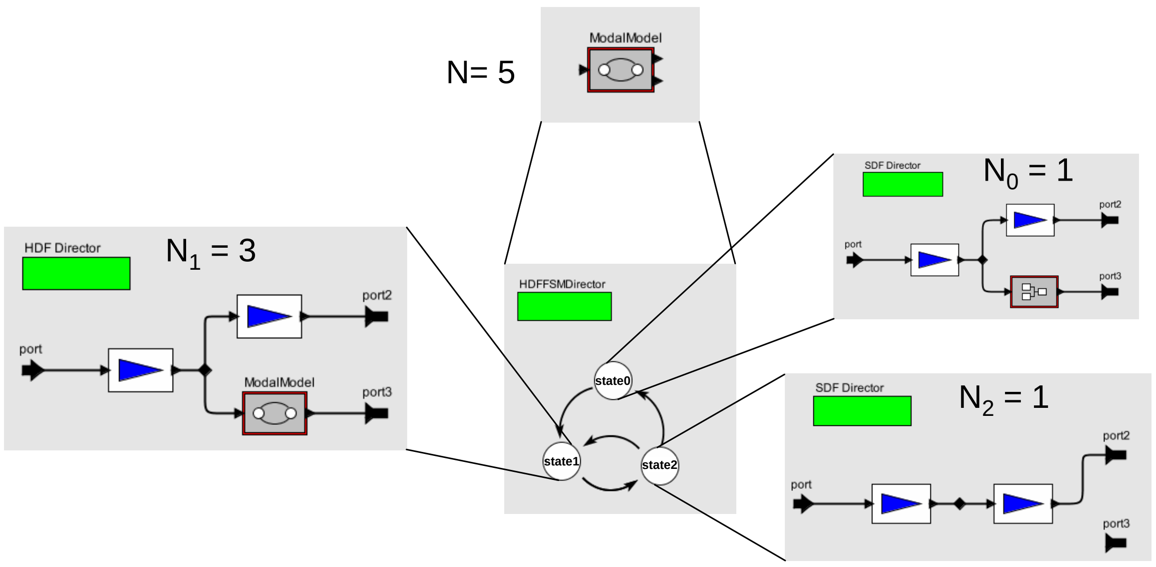


Figure 3.9: Configuration analysis for a modal model.

where k is derived from

$$i - \sum_{j=0}^{k-1} N_j \geq 0$$

and

$$i - \sum_{j=0}^{k} N_j < 0$$

q is derived from

$$q = i - \sum_{j=0}^{k-1} N_j$$

Comments: in HDF, a modal model is controlled by a HDFFSMDirector, which is responsible for making the state transition after one complete iteration at the top level. In Fig. 3.9, the modal model has three states with refinements of 1, 3 and 1 configurations respectively (i.e., $M = 3$, $N_0 = 1$, $N_1 = 3$, $N_2 = 1$). The modal model has "or" semantics, meaning it can be in only one of the states at any time, therefore the total number of configurations for the modal model takes the form of summation—5 in this case (i.e., $N = 5$), and the set of rate signatures for the modal model is the union of the set of rate signatures of each refinement. Assume we want to compute R_3—the signature of the 3rd configuration of the modal model. By substituting $i = 3$ into the above equations, we get $k = 1$ and $q = 2$. That means R_3 is equal to the rate signature of the 2nd configuration ($q = 2$) of the refinement for state1 ($k = 1$).[4] Similarly, R_0 is equal to the rate signature of the 0th configuration of the refinement for state0. R_1 is equal to the rate signature of the 0th configuration of the refinement for state1. And so it goes on.

4) *Composite actor with a local HDFDirector* (see Fig. 3.10):

$$N = \prod_{j=0}^{M-1} N_j$$

where N_j is the number of configurations of the jth contained actor, and M is the number of contained actors.

For $i = 0, \ldots, N-1$,

$$\begin{aligned} R_i \;=\; & \text{the rate signature of the composite actor} \\ & \text{inferred from the local SDF schedule} \end{aligned}$$

when the jth contained actor, for $j = 0, \ldots, M-1$, presents its rate signature in its k_jth configuration, where k_j is derived from

$$i = \sum_{j=0}^{M-1} (k_j \prod_{q=j+1}^{M-1} N_q)$$

[4]all enumerations start from 0.

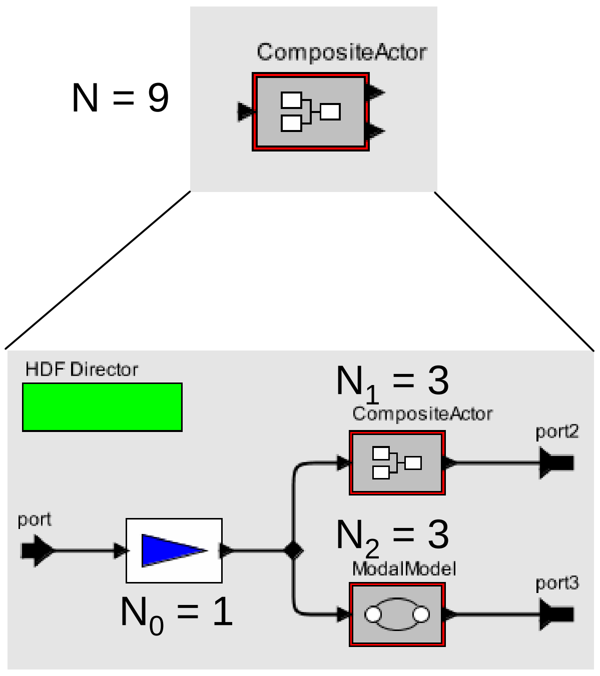


Figure 3.10: Configuration analysis for a composite actor with a local HDFDirector.

$$= \quad k_0 N_1 N_2 \ldots N_{M-1} + k_1 N_2 \ldots N_{M-1} + \ldots + k_{M-2} N_{M-1} + k_{M-1}$$

Comments: In Fig. 3.10, the composite actor has three contained actors with 1, 3 and 3 configurations respectively (i.e., $M = 3$, $N_0 = 1$, $N_1 = 3$, $N_2 = 3$). A composite actor with a local HDFDirector has "and" semantics, meaning the configuration of the composite actor is determined by the configurations of all contained actors together, therefore the total number of configurations for the composte actor takes the form of product—9 in this case (i.e., $N = 9$), and the set of rate signatures for the composite actor are determined by local SDF schedules for all combinations of the rate signatures of all contained actors. Assume we want to compute R_4—the rate signature of the 4th configuration of the composite actor. By substituting $i = 4$ into the above equation, we get $k_0 = 0$, $k_1 = 1$ and $k_2 = 1$. That means R_4 is the rate signature of the composite actor when the 0th contained actor presents its rate signature in its 0th configuration ($k_0 = 0$), the 1st contained actor presents its rate signature in its 1st configuration ($k_1 = 1$) and the 2nd contained actor presents its rate signature in its 1st configuration ($k_2 = 1$). Similarly, R_0 is the rate signature of the composite actor when the 0th contained actor presents its rate signature in its 0th configuration , the 1st contained actor presents its rate signature in its 0th configuration and the 2nd contained actor presents its rate signature in its 0th configuration. R_1 is the rate signature of the composite actor when the 0th contained actor presents its rate signature in its 0th configuration , the 1st contained actor presents its rate signature in its 0th configuration and the 2nd contained actor presents its rate signature in its 1st configuration. And so it goes on.

Looking back at Fig. 3.5, the HDF model has a total of 6 configurations.

3.4 The Complexity Issue

The product form in Fig. 3.10 leads to potential exponential explosion in the number of configurations (thus schedules) for HDF models. Under some circumstances, if a design is done appropriately, a submodel does not change its rate signature presented to the outside when it changes its configuration. In this case, the submodel can be considered an SDF composite actor as far as other outside actors are concerned.

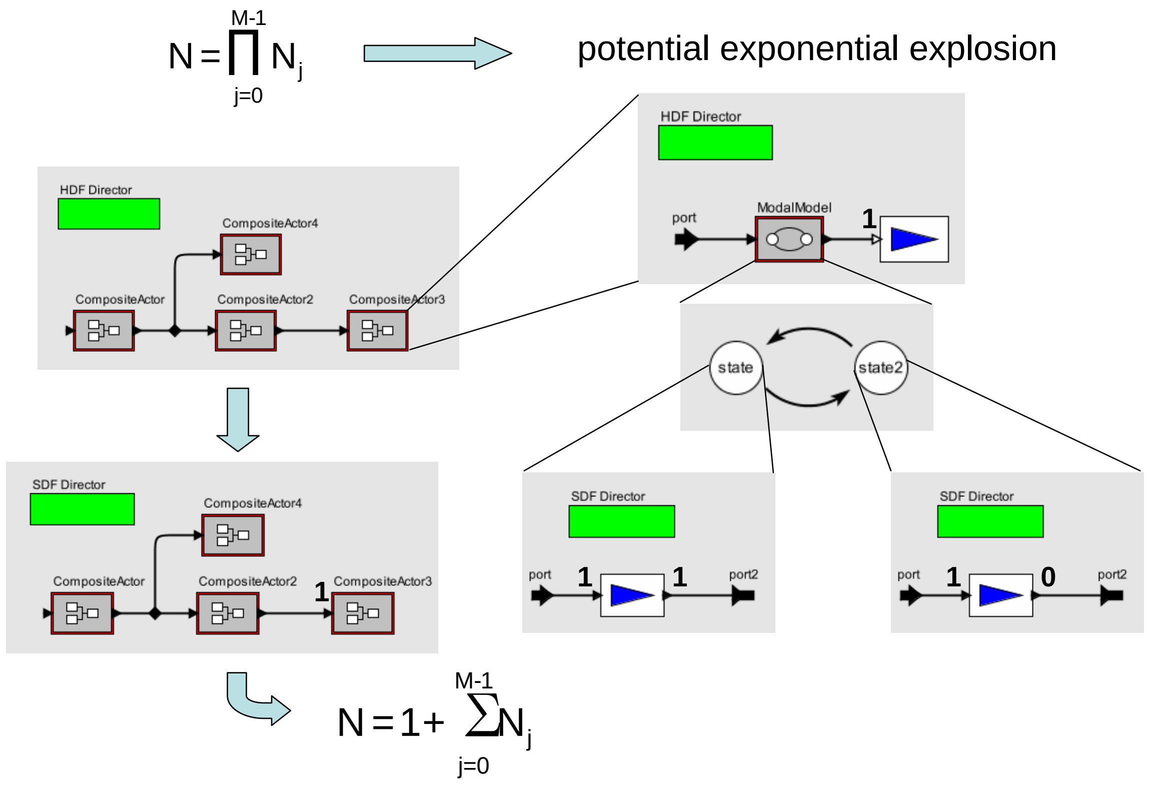


Figure 3.11: The complexity issue in HDF scheduling.

In Fig. 3.11 CompositeActor3 keeps a constant rate of 1 for its input port when its contained modal model makes state transitions. If other composite actors behave the same way, then schedules for the whole model can be decomposed into the schedule for the top level composite actor and the schedules of individual contained composite actors. In another words, the number of schedules required to describe the whole model is reduced from a product form to a sum form.

Chapter 4

Previous Work on Partial Evaluation

Kleene's $s{-}m{-}n$ theorem [Kleene1952] establishes the theoretical foundation for partial evaluation techniques. Formally, the theorem states that for any two integers $m, n \geq 1$, there exists a primitive recursive function S_{mn} that takes as arguments a code p for a function of $m + n$ arguments, and m additional integer values, returns a code for a function of n arguments, and satisfies the following lambda calculus equation for all arguments

$$\varphi^{(n)}_{S_{mn}(p,x_1,\ldots,x_m)} = \lambda y_1, \ldots, y_n.\varphi^{(m+n)}_p(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

where $\varphi^{(k)}_p$ represents a recursive function with k parameters and described by code p, S_{mn} represents the partial evaluator, and $S_{mn}(p, x_1, \ldots, x_m)$ represents the new code after applying the partial evaluator to the original code p and m arguments:

$x_1, \ldots, x_m$.

In practical terms, this means given any programming language and integers $m, n \geq 1$, there is an algorithm with the following property: given the source code for a function with $m + n$ arguments and values for the first m as input, it outputs the source code for a function that effectively hardcodes the first m arguments to those values.

In principle, all partial evaluation techniques are no more than trying to find a way to implement this algorithm. In Kleene's constructive proof of the $s-m-n$ theorem, he's only concerned with the existence of the residual program and efficiency is irrelevant. But in practice, we are more interested in using partial evaluation as optimization techniques and efficiency is the key for the generated residual program. In fact, finding out what part of a program is fixed and finding out the algorithm for doing partial evaluation for the given context and generating highly efficient program, and also the question of how to do it systematically, really depends on a lot factors such as programming languages, applications, and so on—that is the major research problem for a lot of researchers as reviewed in this chapter. And that is also my topic in this thesis, in essence.

4.1 Partial Evaluation in General Purpose Software

4.1.1 Partial Evaluation in Imperative Languages

Partial evaluation was originally formulated for small imperative languages (see [Ershov1977][Bulyonkov1988]). Following that, an off-line partial evaluator for a flow chart language was developed [Gomard1991-2]. A practical partial evaluator for the C programming language that transfered academic results to a realistic context was implemented in [Anderson1994].

The review in this section mostly follows Jones et al [Jones1993]. It uses a very simple flow chart language to explain the basic principles of partial evaluation so that it does not bring up the complexity associated with a real language. In fact the core techniques carry over to more complete imperative languages like C as well as functional and logic languages.

Fig. 4.1 is the syntax of the flow chart language with variables, assignment statements, `goto`s and tests. A program written in this language consists of a number of basic blocks. A label `pp` is attached to each basic block, which consists of a sequence of assignments and a jump statement at the end. A program execution starts with reading the values of input variables. Then the first basic block starts executing. Inside each basic block, the execution is sequential. At the end of each basic block, the execution either jumps to another basic block or returns an expression. In the

⟨Program⟩	::=	`read` ⟨Var⟩, . . . , ⟨Var⟩`;` ⟨BasicBlock⟩$^+$
⟨BasicBlock⟩	::=	⟨Label⟩: ⟨Assignment⟩* ⟨Jump⟩
⟨Assignment⟩	::=	⟨Var⟩ `:=` ⟨Expr⟩`;`
⟨Jump⟩	::=	`goto` ⟨Label⟩`;`
	|	`if` ⟨Expr⟩ `goto` ⟨Label⟩ `else` ⟨Label⟩`;`
	|	`return` ⟨Expr⟩`;`
⟨Expr⟩	::=	⟨Constant⟩
	|	⟨Var⟩
	|	⟨Op⟩ ⟨Expr⟩ . . . ⟨Expr⟩
⟨Constant⟩	::=	`quote` ⟨Val⟩
⟨Op⟩	::=	`hd` | `tl` | `cons` | . . .
		plus any others needed for writing
		interpreters or program specializers
⟨Label⟩	::=	any identifier or number

Figure 4.1: Syntax of a flow chart language [Jones1993].

latter case, the execution ends and the value of the returned expression is the result of the program execution.

Now given a program and only part of this program's input data, a partial evaluator will attempt to execute the given program as far as possible yielding as result a residual program that will perform the rest of the computation when the rest of the input data are supplied. So the question is how to hard-code the partial input data in the residual program.

An imperative language has a notion of state during the program execution. In the flow chart language, the state is a pair `(pp, v)` where `pp` is the label of a basic block representing the current control point during the execution and `v` is the set of program variables holding their current values. The execution of an imperative program is essentially a sequence of state transitions: `(pp, v) => (pp', v')`.

When only part of the input data are available, we can divide the set of input variables into static part and dynamic part (those known are static and those unknown are dynamic). Next we can classify the set of all program variables into static part (`vs`) and dynamic part (`vd`) through a process called *binding-time analysis*. Essentially this analysis determines those variables that only depend on other static variables or constants as static and otherwise as dynamic. Such a classification is called a division. Given a division, the state transitions can be decomposed into `(pp, (vs, vd)) => (pp', (vs', vd'))`. By re-associating, we get `((pp, vs), vd)) => ((pp', vs'), vd')`. In another words, we can incorporate the values of the static variables into the control point during partial evaluation, and the generated residual program is usually more efficient.

Suppose during partial evaluation we discover that if the program had been executed normally with all input data supplied, the computation might eventually be in a state with control at point `pp` and with `vs` as the values of the static variables. Then the pair `(pp, vs)` is made a specialized program point in the residual program. The code that `(pp, vs)` labels in the residual program is an optimized version of the code at `pp` in the original program. The potential for optimization is due to the availability of the values of the static variables. A program point `pp` may appear in several residual versions, each with different values of the static variables. The set of all specialized program points `(pp, vs)` that are reachable during partial evaluation

```
poly := { ($pp_0$, $vs_0$) };
while poly contains an unmarked (pp, vs) do
begin
    mark (pp, vs);
    generate code for the basic block starting at pp using the values in vs;
    poly := poly $\cup$ successors(pp, vs)
end
```

Figure 4.2: A simple partial evaluation algorithm [Jones1993].

is called `poly`. Thus the partial evaluation involves computing `poly` and generating the residual program given `poly`.

Fig. 4.2 gives a simple algorithm for computing `poly` and generating code, where `poly` starts with the first label in the program and the initial values of the static data. For any specialized program point `(pp, vs)` in `poly`, a sequence of assignments is followed by a "passing of control". Some assignments might reassign static variables, thus forcing `vs` to be updated, so a new specialized program point has form `(pp', vs')`. The set of successors naturally depends on the form of control passage and rules for computing successors are given by Fig. 4.3.

The generated code for a basic block is the concatenation of the reduced code for the sequence of assignments and the final jump statement. Fig. 4.4 and Fig. 4.5 show the reduction rule for each statement. They use two helper functions: `eval(exp, vs)`, which returns the value of a static expression `exp`; and `reduce(exp, vs)`, which performs constant folding of static parts of a dynamic expression.

Take the following program as an example

control component of `pp`	*successors(*`pp`, `vs`*)*	
`return`	{}	
`goto pp′`	{`(pp′, vs′)`}	

`if exp`
`goto pp′ else pp″`
	{`(pp′, vs′)`}
	{`(pp″, vs′)`}
	{`(pp′, vs′), (pp″, vs′)`}
		if `exp` evaluates to true
		if `exp` evaluates to false
		if `exp` is dynamic

Figure 4.3: Rules for computing successors [Jones1993].

Command	*Done at specialization time*	*Generated code*
`X := exp` (if `X` is dynamic)	`reduced_exp := reduce(exp, vs)`	`X := reduced_exp`

`X := exp`
(if `X` is static)
	`val := eval(exp, vs);`
	`vs := vs[X ↦ val]`	

`return exp`	`reduced_exp := reduce(exp, vs)`	`return reduced_exp`
`goto pp′`	`goto (pp′, vs)`	

Figure 4.4: Reduction rules for code generation (part I) [Jones1993].

Code generation for a conditional: `if exp goto pp′ else pp″`		
	Done at specialization time	*Generated code*

(if `exp` is dynamic)	`reduced_exp := reduce(exp,vs)`	`if reduced_exp`
		`goto(pp′,vs)`
		`else(pp″,vs)`

(if `exp` is static and `val = true`)	`val := eval(exp,vs)`	`goto (pp′, vs)`
(if `exp` is static and `val = false`)	`val := eval(exp,vs)`	`goto (pp″,vs)`

Figure 4.5: Reduction rules for code generating (part II) [Jones1993].

```
while name != hd (namelist) do
begin
    valuelist := tl (valuelist);
    namelist := tl (namelist)
end;
value := hd (valuelist);
```

which searches the `value` in the `valuelist` for the corresponding `name` in the `namelist`[1]. Suppose the partial evaluator is given the initial values of the variables `name` and `namelist`, for example `name = z` and `namelist = (x y z)` but the `valuelist` is unknown. By converting the above program to a desugared version written in the flow chart language, we get

```
search: if name = hd(namelist) goto found else cont;
cont  : valuelist := tl(valuelist);
        namelist := tl(namelist);
        goto search;
found : value := hd(valuelist);
        return value;
```

Starting with `poly = (search, (z,(x y z)))` and applying the algorithm in Fig. 4.2 and reduction rules in Fig. 4.4 and Fig. 4.5, the following residual program is produced:

```
(search, (z, (x y z))): goto (cont, (z, (x y z)));
(cont,   (z, (x y z))): valuelist := tl (valuelist);
                        goto (search, (z, (y z)));
(search, (z, (y z)))  : goto (cont, (z, (y z)));
(cont,   (z, (y z)))  : valuelist := tl (valuelist);
                        goto (search, (z, (z)));
   (search, (z, (z))) : goto (found, (z, (z)));
    (found, (z, (z))) : value := hd (valuelist);
                        return value;
```

[1]`hd` is the abbreviation for `head` and `tl` is the abbreviation for `tail`.

By applying a technique called *transition compression* which replaces `goto(pp, vs)` with a copy of the corresponding basic block labeled `(pp, vs)`, the residual program becomes

```
(search, (z, (x y z))): valuelist := tl (valuelist);
                        valuelist := tl (valuelist);
                        value := hd (valuelist);
                        return value;
```

For this simple program, one can easily see the equivalence with the original program given the partial input data.

The above description illustrates the basic principles of partial evaluation. In practice a lot of details need to be dealt with, which I summarized below for interested readers. For example, indiscriminate use of transition compression may lead to code duplication and infinite compression; it may be desirable do the compression along with the code generation (i.e., on the fly) to avoid superfluous `goto`s; a technique called *generalization* to classify unbounded static variables as dynamic is needed to avoid non-terminating or useless partial evaluation; in online partial evaluation (the partial evaluation just described is offline), the concrete values computed on the spot can affect the choice of action taken so that more static information can be exploited to yield better residual programs, although at the cost of increased complexity for the partial evaluator; if a dynamic variable only takes finite many values, one can use *the trick* to generate code for each value, thus effectively converting the dynamic variable into a static variable; instead of making a uniform division of static and dynamic variables in the binding-time analysis, *pointwise* divisions (i.e., the division can be different for each basic block) can lead to better result in the generated code; a live variable analysis [Aho1986] can be performed to exclude dead static variables which lead to duplicate code blocks; a *polyvariant* division can assign to each label a set of divisions depending on how the program point is reached while pointwise divisions only depend on the program point. In a real programming language such as C, even more issues need to be considered. Interested readers may read references in this section and even more referred to in those references.

4.1.2 Partial Evaluation in Functional Languages

Many of the ideas and principles that worked for the simple flow chart language in the previous section can be adapted to functional programming languages.[2]

Fig. 4.6 shows a first-order subset of Scheme language. Compared with the flow chart language, there are a lot of similarities that can be taken advantage of during partial evaluation. A flow chart program is a collection of labeled basic blocks; a Scheme program is a collection of named function definitions. In the flow chart program, a program point is the label of a basic block; in a Scheme program, a program point is the name of the defined function. In the flow chart language, values are bound to global, mutable variables and the bindings are created or changed by assignments, and the language has a notion of current state. In Scheme, values are bound to function parameters and the bindings are created by function application

[2]The review in this section also mostly follows Jones et al [Jones1993] although I re-formulated the binding-time analysis in a slightly different way using a fixed-point theorem.

```
⟨Program⟩    ::=  (⟨Equation⟩ ... ⟨Equation⟩)        Function definitions
⟨Equation⟩   ::=  (define (⟨FuncName⟩ ⟨Varlist⟩) ⟨Expr⟩)
⟨Varlist⟩    ::=  ⟨Var⟩ ... ⟨Var⟩                    Formal parameters
⟨Expr⟩       ::=  ⟨Constant⟩                         Constant
               |  ⟨Var⟩                              Variable
               |  (if ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩)          Conditional
               |  (call ⟨FuncName⟩ ⟨Arglist⟩)        Function application
               |  (⟨Op⟩ ⟨Expr⟩ ... ⟨Expr⟩)           Base application
⟨Arglist⟩    ::=  ⟨Expr⟩ ... ⟨Expr⟩                  Argument expressions
⟨Constant⟩   ::=  ⟨Numeral⟩
               |  (quote ⟨Value⟩)
⟨Op⟩         ::=  car | cdr | cons | = | + | ...
```

Figure 4.6: Syntax of a first-order subset of Scheme language [Jones1993].

and cannot be modified; they are immutable and the language has no notion of current state. During program execution, control passes from program point to program point; by jumps in a flow chart program and by function calls in Scheme. During partial evaluation, labeled blocks are specialized with respect to static global variables; similarly, function definitions can be specialized with respect to static function parameters.

In the flow chart language, each global variable is classified as static or dynamic given partial input data; such a classification is called a division. Similarly, in Scheme a division is a classification of each function parameter as static or dynamic, given the values of part of the input parameters of the first function (called the goal function since program execution starts with applying the first function).

Interestingly, the binding time analysis can resort to a fixed-point theorem to find a division. Consider the following program

$$\texttt{(define (}\mathtt{f_1}\ \mathtt{x_{11}}\ \ldots\ \mathtt{x_{1a_1}}\texttt{)}\ \mathtt{e_1}\texttt{)}$$

$$\vdots$$

$$\texttt{(define (}\mathtt{f_n}\ \mathtt{x_{n1}}\ \ldots\ \mathtt{x_{na_n}}\texttt{)}\ \mathtt{e_n}\texttt{)}.$$

A division is a mapping from all function parameters to the abstract values $\{\mathtt{S},\ \mathtt{D}\}$:

$$\mathtt{div} : \{\mathtt{x_{11}}\ \ldots\ \mathtt{x_{1a_1}}\ \ldots\ \mathtt{x_{n1}}\ \ldots\ \mathtt{x_{na_n}}\} \rightarrow \{\mathtt{S},\ \mathtt{D}\}.$$

By introducing a partial order on the set $\{\mathtt{S},\ \mathtt{D}\} : \mathtt{S} < \mathtt{D}$, we can induce a partial order on the set of all divisions:

$$\mathtt{div_1} \leq \mathtt{div_2} \Leftrightarrow \mathtt{div_1(x)} \leq \mathtt{div_2(x)}\ \texttt{for any input parameter x}.$$

We can define a function $\mathtt{F} : \{\mathtt{div}\} \rightarrow \{\mathtt{div}\}$ the following way: given a division, for any function $\mathtt{f_i}, \mathtt{i} = \mathtt{1} \ldots \mathtt{n}$, find all applications of $\mathtt{f_i}$ in $\mathtt{e_1}, \ldots, \mathtt{e_n}$. For each application of $\mathtt{f_i}$, we can decide a new division for $\mathtt{f_i}$'s input parameters: for each input parameter, if the corresponding expression used in the application contains any dynamic parameter in the original division, then this input parameter is dynamic in the new division; otherwise it is static. Then we take the least upper bound (LUB) of all new divisions for the input parameters of $\mathtt{f_i}$ as the result of applying $\mathtt{F}$ to the original division.[3] It can be shown that the function $\mathtt{F}$ thus defined is monotonic. Given a least division $\mathtt{div_0}$ which classifies the input parameters for the first function

[3]Strictly, the result should be projected onto the the input parameters of $\mathtt{f_i}$.

as pre-given and the rest as static, F is a continuous function on a CPO with the least element $\texttt{div}_0$ as there are only finite many divisions. By using the fixed-point theorem, F has a least fixed point, which can be derived by applying F repetitively, starting with $\texttt{div}_0$, for a finite number of times. The least solution satisfies our goal of finding a least dynamic solution (thus more opportunity for partial evaluation) to the fixed-point equation.

The partial evaluation algorithm for Scheme is very similar to that for flow charts. It has a set pending of functions yet to be specialized, and a set marked of those already specialized. As long as pending is non-empty, it repeatedly selects and removes a member (f, vs) from pending, and constructs a version of function f, specialized to the values vs of its static parameters. The specialization of f's body with respect to vs may require new functions to be added to pending, namely those called from the specialized body. As in the flow chart partial evaluator, the residual program is complete when pending is empty. A Scheme function body is specialized by reducing it symbolically, using the values of static variables.

Like the previous section, this one only touches the basics of partial evaluation for functional languages. Interested readers may refer to [Bondorf1990][Bondorf1991] [Bondorf1992][Bondorf1991-2] for more details, including a partial evaluator called Similix, which can deal with high-order functions and side effects.

4.1.3 Partial Evaluation in Object-Oriented Language

Object-oriented programming (OOP) has become more commonly used in mainstream software application development, as a way to manage the ever increasing complexity of modern software projects. OOP uses encapsulation to hide implementation details of a class from other objects and uses abstraction to build generic, reusable components with techniques such as ever popular design patterns [Gamma1994].

However, OOP achieves flexibility and reusability—the gold standard of good programming—at the expense of efficiency. Encapsulation isolates individual program components and increases the cost of data access. Method invocation is implemented using virtual dispatching/late binding, resulting in run-time overhead. Moreover, it obscures program control flow and blocks traditional hardware and software optimizations. Compiler optimization techniques for OOP have advanced considerably and can eliminate some of these overheads. However, compared with partial evaluation techniques, compilers apply optimizations in a more restricted manner, with the goal of producing a program that performs well in a *normal* usage context. Partial evaluators are more aggressive in propagating inferred information. They propagate values of any type, including partially known objects, throughout the program and reduce any computation that is based solely on known information. Thus a partial evaluator can achieve more pervasive optimization than a typical compiler [Schultz2000].

The execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Specifying particular parts

```
abstract class Binary {
    abstract int eval(int x,int y);
    abstract int neutral();
}
class Add extends Binary {
    int eval(int x,int y) {
        return x+y;
    }
    int neutral() { return 0; }
}
class Mult extends Binary {
    int eval(int x,int y) {
        return x*y;
    }
    int neutral() { return 1; }
}
```

```
class Power {
    int exp; Binary op;
    Power(int exp,Binary op) {
        this.exp = exp;
        this.op = op;
    }
    int raise(int base) {
        int result = op.neutral();
        int e = exp;
        while( e-- > 0 )
            result = op.eval(result,base);
        return result;
    }
}
```

Figure 4.7: Binary operators and a power function [Schultz2003].

of the program context can fix certain parts of this interaction. Partial evaluation simplifies the object interaction by evaluating the static interactions, leaving behind only the dynamic interactions.

Take the favorite example in [Schultz2003], which uses a strategy design pattern. Fig. 4.7 shows a collection of four classes: Binary, Add, Mult, and Power. The abstract class Binary is the superclass of the two concrete binary operators Add and Mult (the strategies). The Power class can be used to apply a Binary operator a number of times to a base value, as illustrated by the following expression:

```
(new Power(3, new Mult())).raise(x)
```

which computes x^3. The object interaction of this program is shown on the left side of Fig. 4.8.

Invoking the method raise(x) of the Power object gives rise to a series of object

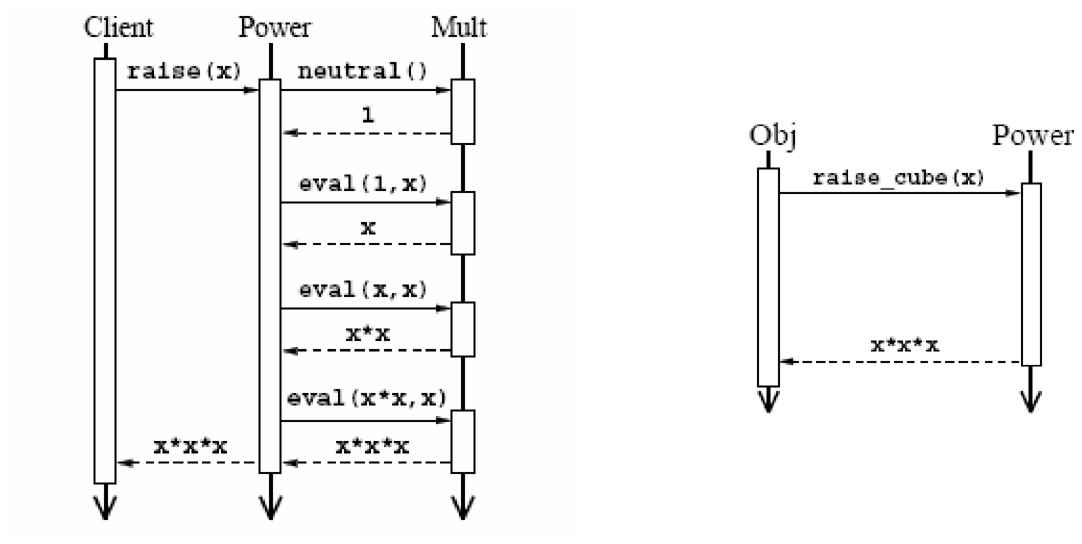


Figure 4.8: Object interactions before and after partial evaluation [Schultz2003].

interactions between the Power object and the Mult object that results in the return value x*x*x. To optimize for this case, we can enable the Power object to produce the result x*x*x directly. Specifically, we can add a method

```
int raise_cube(int base) {
    return base * base * base;
}
```

to the class Power, and clients can use this new method to compute the result more efficiently. The resulting interaction is shown on the right side of Fig. 4.8. Partial evaluation can derive such a specialized method automatically, using constant propagation, loop unrolling, and virtual function call elimination. Some of these techniques are derived directly from imperative languages.

4.2 Partial Evaluation in Embedded Software

The use of partial evaluation in embedded software has been more recent. Embedded software is usually highly domain-specific and has its own idioms/patterns for expressing a design, therefore a partial evaluator can expand its techniques to take advantage of this special knowledge. After all, the spirit of partial evaluation is that the more that is known, the more it can do to generate efficient program.

One example is the Click component framework for PC router construction (see [Kohler2002]). Click achieves its flexibility and extensibility by building routers and other packet processors from fine-grained packet processing modules called elements (see Fig. 4.9). Even though component systems make networking software easier to program, component techniques introduce inefficiencies that monolithic software can avoid, as networks get faster at an even greater rate than processors, making the efficiency of networking software ever more important.

One solution is to apply partial evaluation techniques at the level of networking components. The Click optimization kits include tools called click-fastclassifier, click-devirtualize, click-xform, click-undead, etc.

For example, classifiers are generic elements for classifying packets based on a decision tree built from textual specifications. The click-fastclassifier tool would generate new source code for a classifier based on the specific decision tree and replace the generic element with this more specific element. The click-devirtualize tool addresses virtual function call overhead. It changes packet-transfer virtual function calls

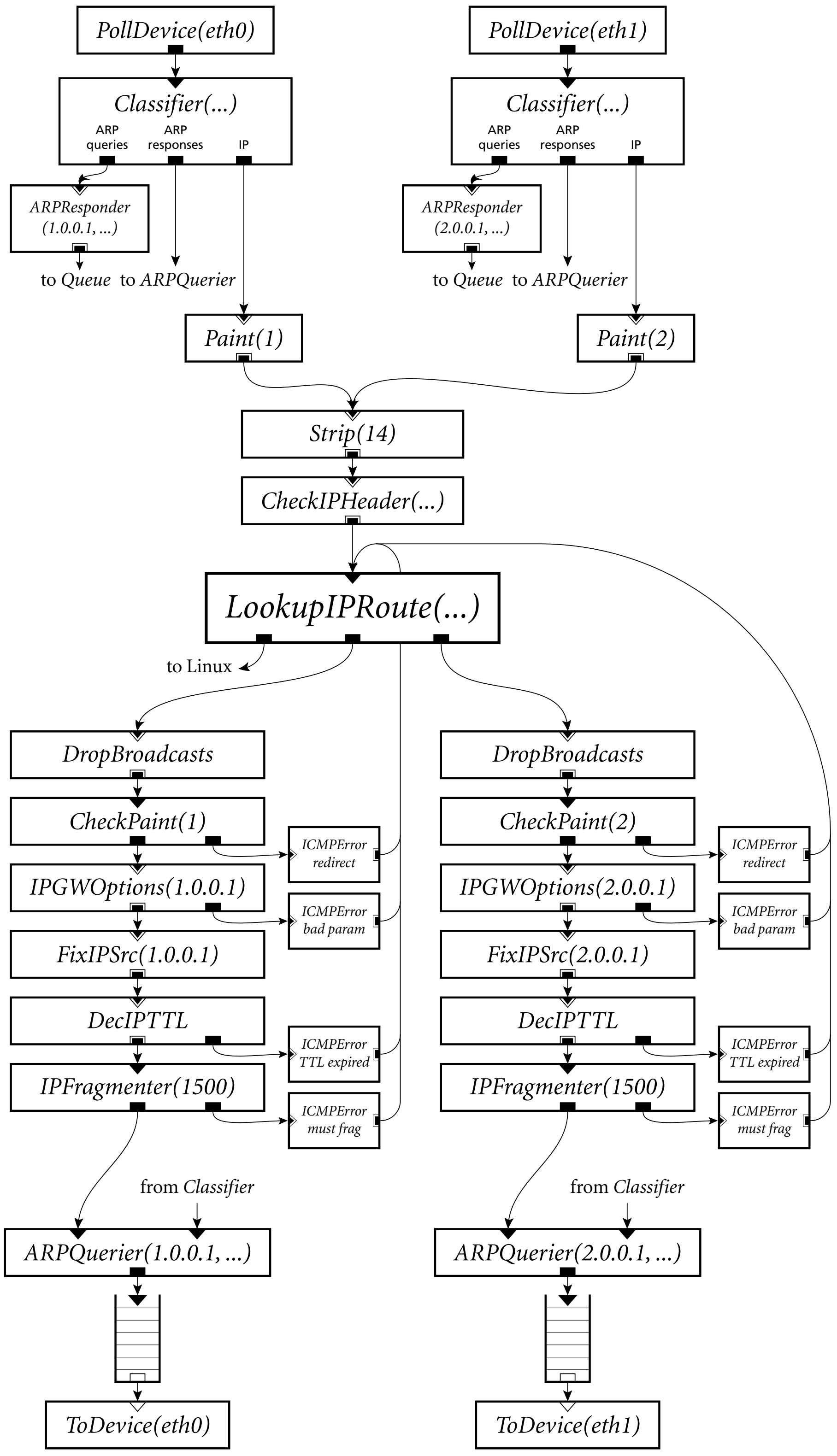


Figure 4.9: A Click IP router built by composing elements [Kohler2002].

into conventional function calls by finding the downstream component and explicitly calling the method on that component. Again this involves transforming the source code so that method binding can be done in the compile time. The click-xform tool reads a router configuration and a collection of pattern and replacement subgraphs. It replaces general-purpose element collection with the corresponding combination element through pattern matching. It thus lowers function call costs by reducing the number of elements in a forwarding path, and reduces the overhead of general-purpose code. All these tools are partial evaluation techniques taking advantage of the knowledge in a specific configuration.

The Koala component model for consumer electronics software [Ommerling2000] is another example of applying partial evaluation for generating more efficient implementation.

Chapter 5

Code Generation for Actor-Oriented Design with Partial Evaluation

5.1 Related Work

There have been a few design frameworks with code generation functionality. Simulink with Real-Time Workshop (RTW), from the Mathworks, is probably in the most widespread use as a commercial product [Simulink]. It can automatically generate C code from a Simulink model and is quite innovative in leveraging an underlying preemptive priority-driven multitasking operating system to deliver real-time behavior based on rate-monotonic scheduling. However, like most design frameworks, Simulink defines a fixed MoC: continuous time is the underlying MoC for Simulink, with discrete time treated as a special case. It can be integrated with another Mathworks product called Stateflow, used to describe complex logic for event-driven systems, for simulation and code generation. The platform I worked on, Ptolemy II, is a software lab for experimenting with multiple concurrency formalisms for embedded system design. It does not have a built-in MoC. The code generation framework built on Ptolemy II is flexible and extensible. It is capable of generating code for multiple MoC's. In particular, I am most interested in code generation for the MoC's whose schedulability is decidable.

There has been previous work on code generation for the Ptolemy II system [Tsay2000]. The approach there involves transformation of the existing source code (i.e. Java code) in each actor of a model, which results in simplified and hence more efficient Java code. Then a generic Java-to-C converter is used to produce compilable C code. The generated code is not efficient enough though. Following a similar approach, Neuendorffer created a component-specialization framework, codenamed Copernicus [Neuendorffer2004], to transform a Ptolemy model into self-contained Java code for efficient execution. The advantage of the Copernicus approach is that it uses the same Java code as used in simulation for code generation and thus guarantees the original model designed and the final code generated are semantically equivalent (if code generation is performed correctly).

There are several ways to execute generated self-contained Java code, either through Java Real Machine or Java Virtual Machine. A Java Real Machine processes bytecodes (JVM instructions) directly in hardware. Sun Microsystems initially proved the concept in the late 1990s with its PicoJava chip. Since then several independent Java hardware implementations have hit the market. They come in two different configurations. Chips of the first type, such as Nazomi Communications' JA108 Multimedia Application Processor, operate as Java coprocessors in conjunction with a general-purpose microprocessor, in much the same way that graphics accelerators are used. Java chips in the other category, such as Imsys's Cjip Processor, aJile's aJ-100 and Parallax's Javelin Stamp Interpreter Chip, replace the general-purpose CPU. It will be interesting to see how this market will develop.

However, the majority of today's embedded microprocessors have their own instruction set. They are either programmed with assembly languages or high level programming languages, mostly C. In order to execute Java code, a Java Virtual Machine has to be ported to these platforms. Even with efficient Java code generated from Copernicus, it still suffers inefficiency inherent in the bytecode interpreter in the JVM. JIT (Just-In-Time) compilers, which compile bytecode on the fly during execution, generally aren't suitable for embedded applications. They produce good performance improvements in desktop Java applications, but the large memory requirement places JIT compilers out of reach for many categories of embedded applications. Even if these speedup techniques can be applied, there is still the critical obstacle to many embedded applications requiring real-time processing due to the

Garbage Collector (GC). The GC thread works in the background to manage the allocation of memory, and the resulting uncertainty as to when the GC will run and for how long is unacceptable. The Real Time Specification for Java (RTSJ) was created to specifically deal with this problem [Bollella2000][Sun][Timesys]. In particular, the Metronome project's real-time garbage collection technology provides sub-millisecond worst-case latency and deterministic scheduling with guaranteed worst-case utilization [Bacon2005]. The work along this path has potential, but it remains to be seen whether it will have a large impact in the area of embedded applications.

In this thesis I take an alternative and more aggressive approach. The goal is to generate code with similar performance as handwritten code so that the benefit offered by the high level abstraction comes with no performance penalty [Zhou2007].

5.2 Overview of Code Generation Framework

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

The code generation framework takes a model that has been shown to meet certain design specifications through simulation and/or verification. Through model analysis—the counterpart of binding-time analysis in the traditional use of partial evaluation for general purpose software, it can discover the execution context for the model and the components contained within. It then generates the target code specific to the execution context while preserving the semantics of the original model. See Fig. 5.1, which follows notions used in [Jones1993].

In this thesis, C is the primary target language. In the generated target code, the variables representing the buffers in the input ports of each actor are defined with the data types discovered through type resolution. At the same time, if the model has a static schedule, then buffer sizes can be predetermined and defined too (as arrays), thus eliminating the overhead of dynamic memory allocation. Through model analysis, the framework can also classify parameters into either static or dynamic. Static parameters have their values configured by users and stay constant during execution. Therefore there is no need to allocate memory for them and every time a static parameter gets used in the generated code, it gets evaluated at the code generation time. On the other hand, dynamic parameters change their values during execution. Therefore a corresponding variable is defined for each of them in the generated code.

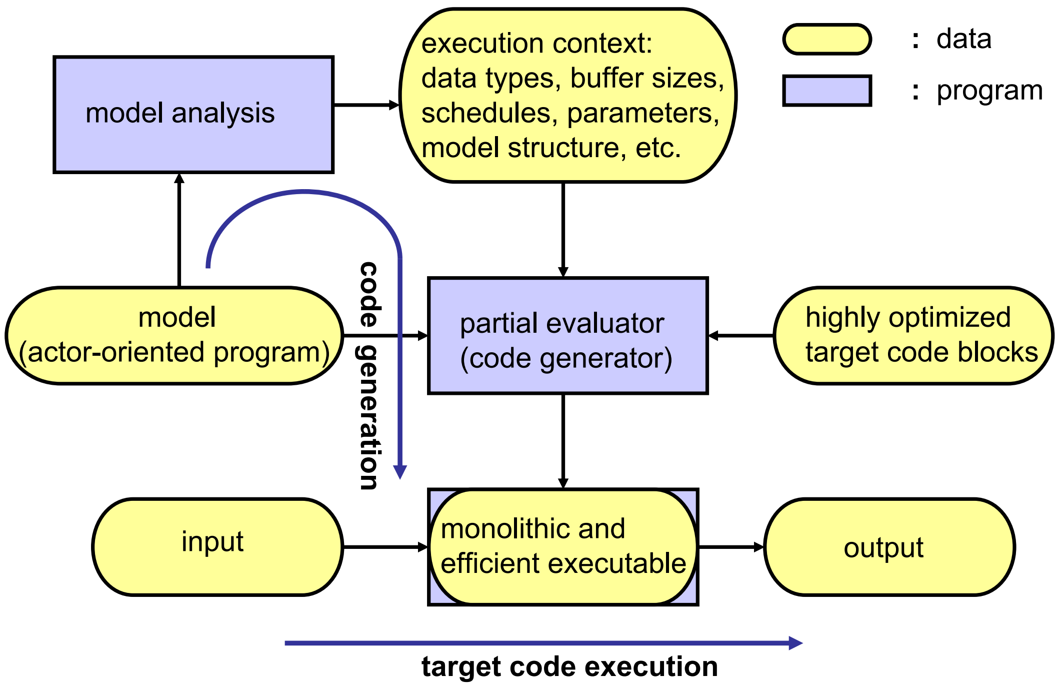

model analysis
execution context: data types, buffer sizes, schedules, parameters, model structure, etc.
: data
: program
code generation
model (actor-oriented program)
partial evaluator (code generator)
highly optimized target code blocks
input
monolithic and efficient executable
output
target code execution

Figure 5.1: Code generation with partial evaluation for actor-oriented programs.

Most of models have static structures. The code generation framework takes advantage of this and eliminates the interfaces between components. In the generated code, instead of using a dozen or so indirection function calls to transfer data between components, a simple assignment is used, resulting in very efficient execution. For the MoC's that have static schedules, instead of dispatching actors based on the schedule, the schedule is hard-coded into the generated code, i.e., the code flow directly reflects the execution sequence, thus making it run much faster. Finally, for each actor that supports code generation, there is a corresponding helper which reads in pre-existing code blocks written in the target language. These target code blocks are functionally equivalent to the actor written in Java, the language used for Ptolemy II. The helper mechanism is elaborated in the next section.

5.3 A Helper-based Mechanism

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

To achieve readability and maintainability in the implementation of helpers, the target code blocks (for example, the initialize block, fire block, and wrapup block) of each helper are placed in a separate file under the same directory. So a helper essentially consists of two files: a java class file and a code template file. This not only decouples the writing of Java code and target code (otherwise the target code would be wrapped in strings and interspersed with java code), but also allows using a target language specific editor while working on the target code blocks. For example, in the Eclipse Integrated Development Environment, the C/C++ Development Toolkit (CDT) provides C and C++ extensions to the Eclipse workbench as a set of Eclipse plug-ins. The convenient features such as keyword highlights in the C/C++ specific editor could help the writing of C code, resulting in improved productivity.

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

5.3.1 What is in a C Code Template File?

A C code template file has a .c file extension but it is not C-compilable due to its unique structure. A CodeStream class is implemented to parse and use these files. The following are the C code template files for the Pulse actor and CountTrues actor.

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

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

// CountTrues.c

/*** preinitBlock ***/
int $actorSymbol(trueCount);
int $actorSymbol(i);
```

```
/**/

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

A C code template file consists of one or more C code blocks. Each code block has a header and a footer. The header and footer tags serve as code block separators. The footer is simply the tag "/**/". The header starts with the tag "/***" and ends with the tag "***/". Between the header tags are the code block name and optionally an argument list. The argument list is enclosed by a pair of parentheses "()" and multiple arguments in the list are separated by commas ",". A code block may have arbitrary number of arguments. Each argument is prefixed by the dollar sign "\$" (e.g., \$value, \$width), which allows easy searching of the argument in the body of code blocks, followed by straight text substitution with the string value of the argument. Formally, the signature of a code block is defined as the pair (N, p) where N is the code block name and p is the number of arguments. A code block (N, p) may be overloaded by another code block (N, p') where $p \neq p'$.[1] Furthermore, different helpers in a class hierarchy may contain code blocks with the same (N, p).

[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.

So a unique reference to a code block signature is the tuple (H, N, p) where H is the name of the helper.

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

5.3.2 What is in a Helper Java Class File?

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

5.3.3 The Macro Language

In the generated code, the input and output ports no longer hold tokens, but instead correspond to data memory where token values will be stored (for example, global variables in C code generation). A helper can also define new variables as needed. A helper, however, does not have the knowledge of the context where these code blocks will be used. Therefore instance-specific information such as the width of actor ports or the full variable names can only be resolved during the code generation time. For helper writers, a set of macros are provided for this purpose.

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

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

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

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

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

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

5.3.4 The CountTrues Example

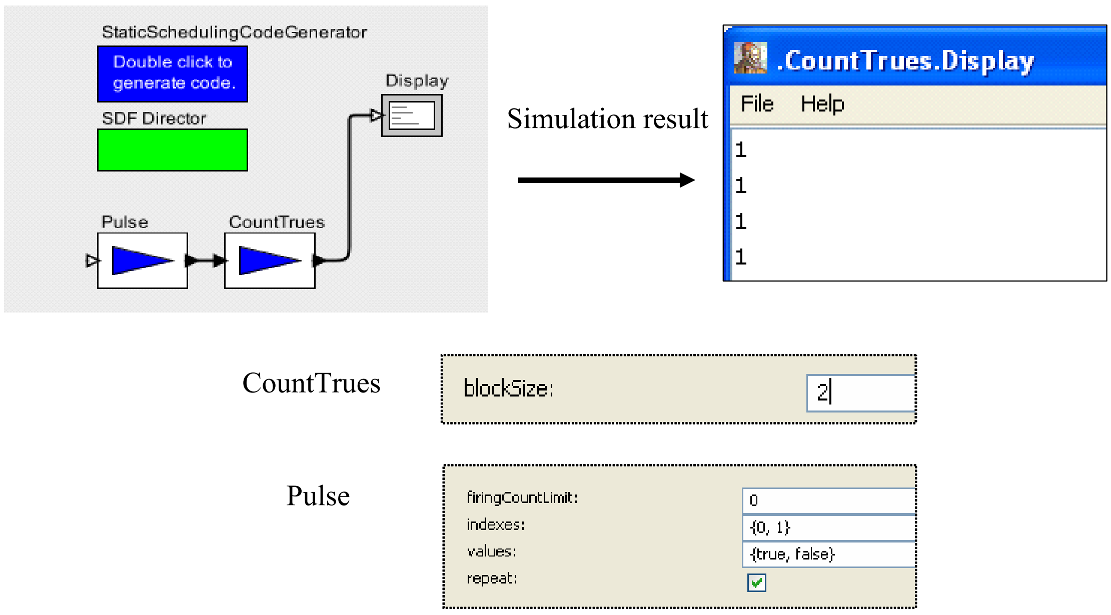
Fig. 5.2 shows a very simple model named CountTrues (notice the name of the model is the same as that of the CountTrues actor used in the model) in the synchronous dataflow (SDF) domain (In Ptolemy II, a domain realizes an MoC). Each time the Pulse actor fires, it produces one token with the value "true" or "false" and the output pattern is determined by its parameters. Each time the CountTrues actor fires, it consumes a certain number of tokens (specified by its "blockSize" parameter) and counts how many "true" tokens are consumed during this firing and then produces one token with the counting value. The Display actor consumes one token and displays the value of the token in each firing. For this simple model, we can easily determine the number of firings for each actor in one periodic firing schedule of the model: f_{Pulse} = blockSize, $f_{CountTrues} = 1$, $f_{Display} = 1$. When the model is simulated in the Ptolemy II framework, the produced result is shown on the right hand side of Fig. 5.2 (the model is fired 4 times because the SDFDirector's "iterations" parameter is set to 4). The following is the main function of the generated stand-alone C program.

```
......
static int iteration = 0;

main(int argc, char *argv[]) {

    init();

    /* Static schedule: */
    for (iteration = 0; iteration < 4; iteration ++) {
```



```
1
1
1
1
```

CountTrues

blockSize: 2

Pulse

firingCountLimit:	0
indexes:	{0, 1}
values:	{true, false}
repeat:	✓

Figure 5.2: The CountTrues model and its simulation result.

```
/* fire Composite Actor CountTrues */
/* fire Pulse */
if (_CountTrues_Pulse_indexColCount < 2
        && _CountTrues_Pulse_iterationCount ==
        Array_get(_CountTrues_Pulse_indexes_ ,
        _CountTrues_Pulse_indexColCount).payload.Int) {
    _CountTrues_CountTrues_input[0] =
    Array_get(_CountTrues_Pulse_values_ ,
        _CountTrues_Pulse_indexColCount).payload.Boolean;
    _CountTrues_Pulse_match = 1;
} else {
    _CountTrues_CountTrues_input[0] = 0;
}

if (_CountTrues_Pulse_iterationCount
        <= Array_get(_CountTrues_Pulse_indexes_ ,
        2 - 1).payload.Int) {
    _CountTrues_Pulse_iterationCount ++;
}
if (_CountTrues_Pulse_match) {
    _CountTrues_Pulse_indexColCount ++;
    _CountTrues_Pulse_match = 0;
}
if (_CountTrues_Pulse_indexColCount >= 2 && true) {
    _CountTrues_Pulse_iterationCount = 0;
    _CountTrues_Pulse_indexColCount = 0;
}

/* fire Pulse */
// The code for the second firing of the Pulse actor is
// omitted here.
.....
.....

/* fire CountTrues */
_CountTrues_CountTrues_trueCount = 0;

for(_CountTrues_CountTrues_i = 0; _CountTrues_CountTrues_i < 2;
        _CountTrues_CountTrues_i++){
    if (_CountTrues_CountTrues_input[(0 +
    _CountTrues_CountTrues_i)%2]) {
        _CountTrues_CountTrues_trueCount++;
```

```
            }
        }
        _CountTrues_Display_input[0] = _CountTrues_CountTrues_trueCount;

        /* fire Display */
        printf("Display: %d\n", _CountTrues_Display_input[0]);
    }

    wrapup();

    exit(0);
}
```

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

```
Display: 1
Display: 1
Display: 1
Display: 1
```

5.4 Software Infrastructure

My code generation framework has the flavor of CG (i.e., Code Generation) domains in Ptolemy Classic [Pino1995]. However, in Ptolemy Classic, code generation domains and simulation domains are separate and so are the actors (called stars in Ptolemy Classic terminology) used in these domains. In Ptolemy Classic, the actors in the simulation domains participate in simulation whereas the actors in the code generation domains participate in code generation. Separate domains (simulation vs. code generation) make it hard to integrate the model design phase with the code generation phase and streamline the whole process. Separate actor libraries make it difficult to maintain consistent interfaces between simulation actors and code generation actors.

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

In addition, my code generation framework takes advantage of new technologies developed in Ptolemy II such as the truly polymorphic type system, richer variety of MoCs including hierarchical concurrent finite-state machines [Girault1999] which are well suited for embedded system design.

To gain an insight into the code generation software infrastructure, it is worthwhile to take a look at how actors are implemented for simulation purposes. Fig. 5.3 shows a simplified UML diagram of key classes to support execution (i.e., model simulation)

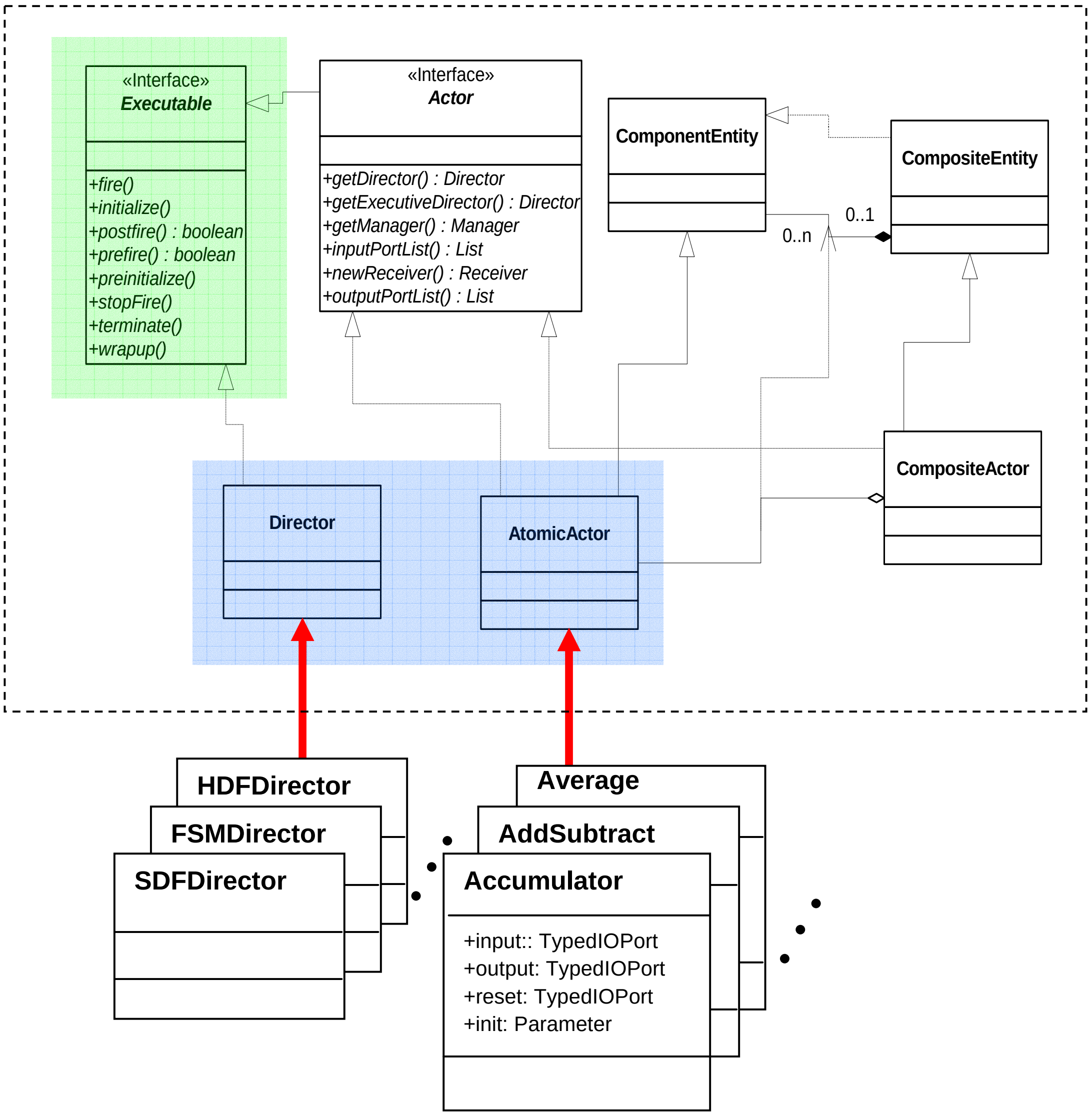


Figure 5.3: Key classes to support execution in Ptolemy II.

in the ptolemy.actor package. (ComponentEntity and CompositeEntity are located in the ptolemy.kernel package. They are included here to give a global picture of class inheritance relations.) As one can see, the Executable interface[2] defines how an actor can be invoked. The preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution. The initialize() method is assumed to be invoked once after the type resolution. It may be invoked again to re-initialize a (sub)model. For example, when a modal model makes a transition and the reset parameter of the transition is true, the submodel associated with the new state is re-initialized. The prefire(), fire(), and postfire() methods are usually invoked many times, with each sequence of method invocations defined as one iteration. The wrapup() method is invoked exactly once per execution at the end of the execution.

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

Fig. 5.4 shows a simplified UML diagram of key classes to support code generation, located in the subpackages under ptolemy.codegen (In Ptolemy II architecture, all the

[2]One must distinguish the use of the term *interface*. Here *interface* is a Java terminology used in object-oriented design. Previously *interface* in component interface is a terminology used in actor-oriented design. It is unfortunate the term is used in different places to have different meaning. But its use is well-established in each field and one can usually tell from its context.

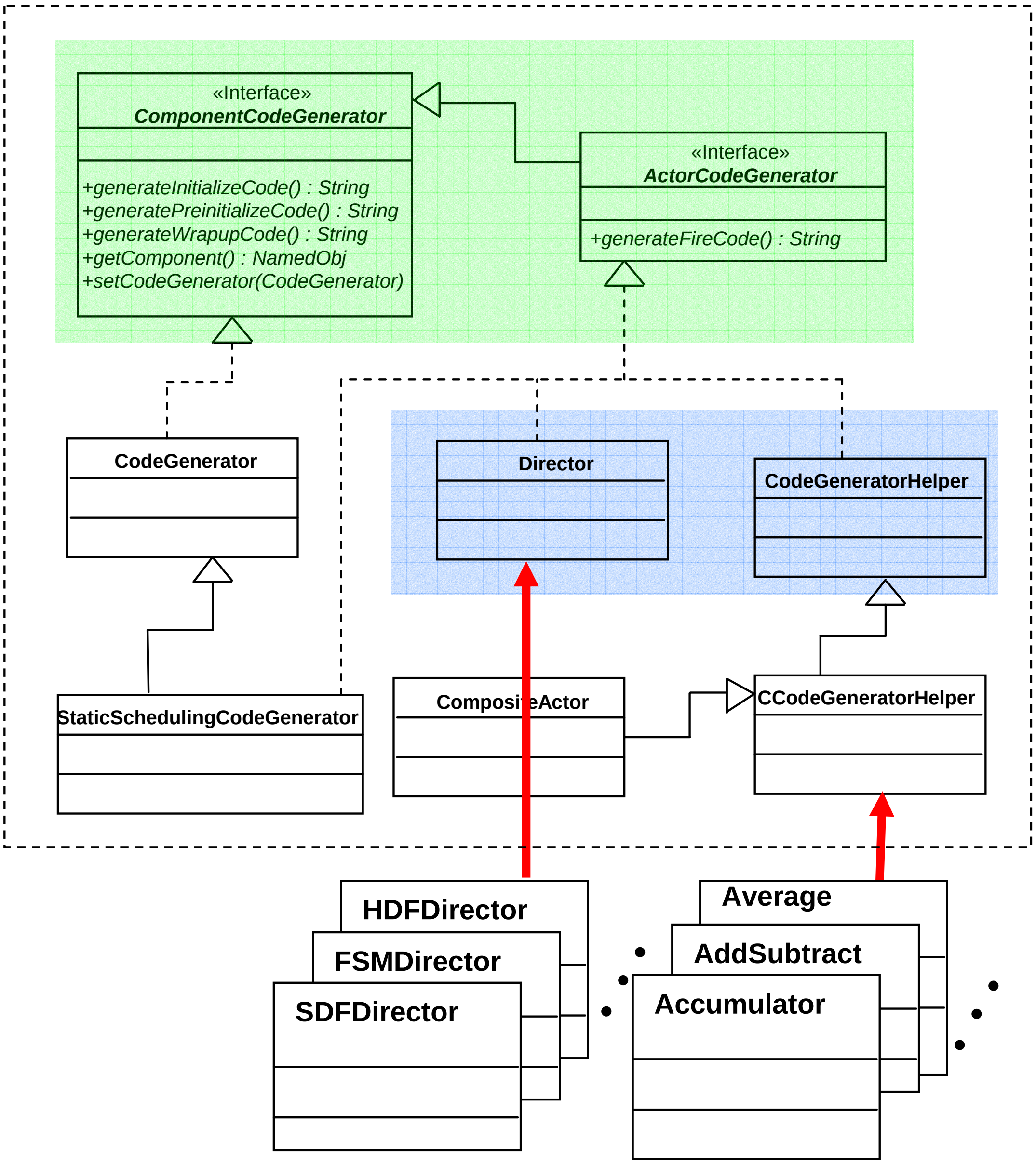

«Interface»
ComponentCodeGenerator
+generateInitializeCode() : String
+generatePreinitializeCode() : String
+generateWrapupCode() : String
+getComponent() : NamedObj
+setCodeGenerator(CodeGenerator)
«Interface»
ActorCodeGenerator
+generateFireCode() : String
CodeGenerator
Director
CodeGeneratorHelper
StaticSchedulingCodeGenerator
CompositeActor
CCodeGeneratorHelper
HDFDirector
FSMDirector
SDFDirector
Average
AddSubtract
Accumulator

Figure 5.4: Key classes to support code generation in Ptolemy II.

package paths start with “ptolemy”). As one can see, the helper class hierarchy and package structure mimic those of regular Ptolemy II actors. The counterpart of the Executable interface is the ActorCodeGenerator interface. This interface defines the methods for generating target code in different stages corresponding to what happens in the simulation. These methods include generatePreinitializeCode(), generateInitializeCode(), generateFireCode(), generateWrapupCode(), etc.

CodeGeneratorHelper, the counterpart of AtomicActor, is the base class implementing the ActorCodeGenerator interface and provides common functionality for all actor helpers. Actors and their helpers not only have the same class hierarchy and package structure, but the same names so that the Java reflection mechanism can be used to load the helper for the corresponding actor during the code generation. For example, there is a Ramp actor in the package ptolemy.actor.lib. Correspondingly, there is a Ramp helper in the package ptolemy.codegen.c.actor.lib (see Fig. 5.5). Here c represents the fact that all the helpers under ptolemy.codegen.c generate C code. Assume we would like to generate code for another target language X, the helpers for generating X code could be implemented under ptolemy.codegen.x. This would result in extensible code generation framework. Developers could not only contribute their own actors and helpers, but also extend the framework to generate code for a new target language.

To generate code for hierarchically composed models, helpers for composite actors are created. The most commonly used composite actor is TypedCompositeActor in

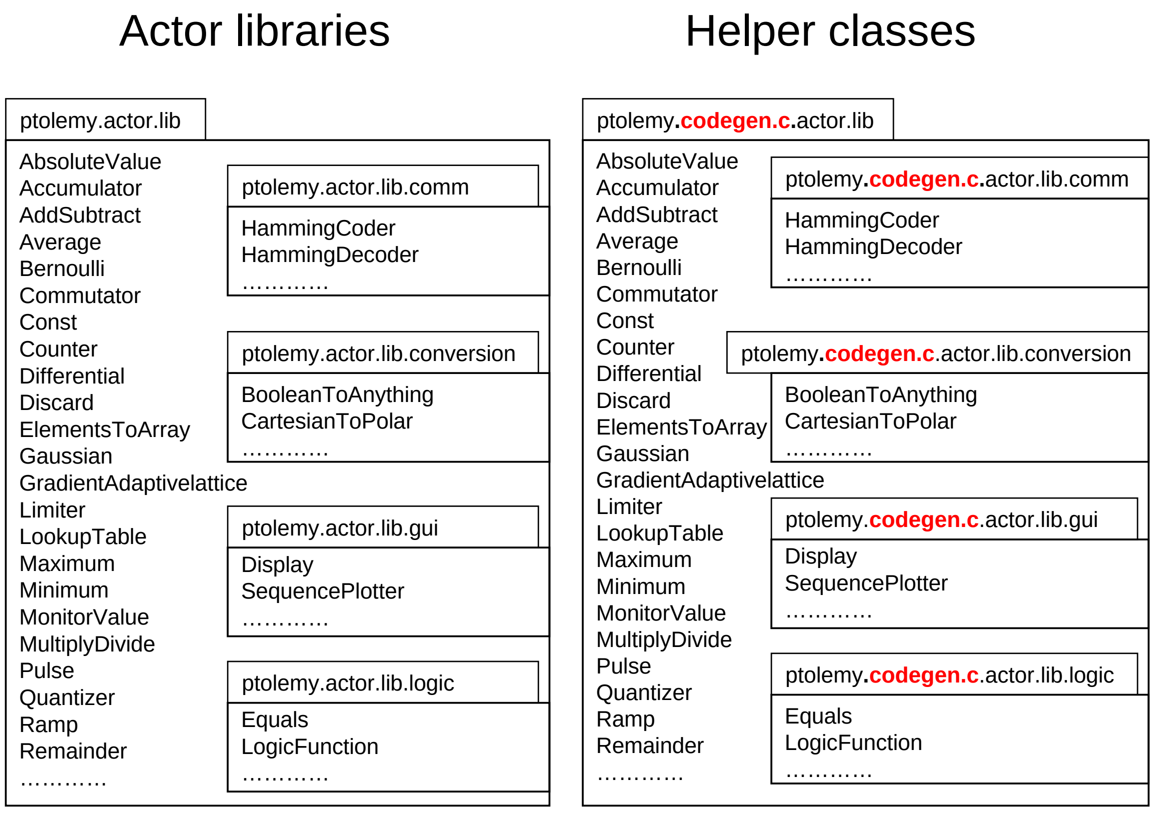


Figure 5.5: Actor libraries and corresponding helpers.

the package ptolemy.actor. A helper with the same name is created in the package ptolemy.codegen.c.actor. The main functionality of this helper is to generate code for the data transfer through the composite actor's interface and delegate the code generation for the composite actor to the helper for the local director or the helpers for the actors contained by the composite actor. Other composite actors include ModalModel, Refinement, etc. and corresponding helper is created for each of them. These composite actors and helpers are used in code generation for modal models.

Since a director implements an MoC (or a domain in Ptolemy II terminology), a helper is created for each director that supports code generation (see Fig. 5.6). These director helpers generate target code that preserves the semantics of MoC's. Currently, the synchronous dataflow domain (SDF), finite state machines (FSM), and heterochronous dataflow domain (HDF) support code generation. More details will follow in the next section.

Finally the StaticSchedulingCodeGenerator class is used to orchestrate the whole code generation process. An instance of this class is contained by the top level composite actor (represented by the blue rectangle in Fig. 5.2). The code generation starts at the top level composite actor and the code for the whole model is generated hierarchically, much similar to how a model is simulated in Ptolemy II environment.

The flow chart in Fig. 5.7 shows each step of the code generation process. The details of some steps are MoC-specific. Notice that the steps outlined in the figure do not necessarily follow the order in the final code. For example, only dynamic

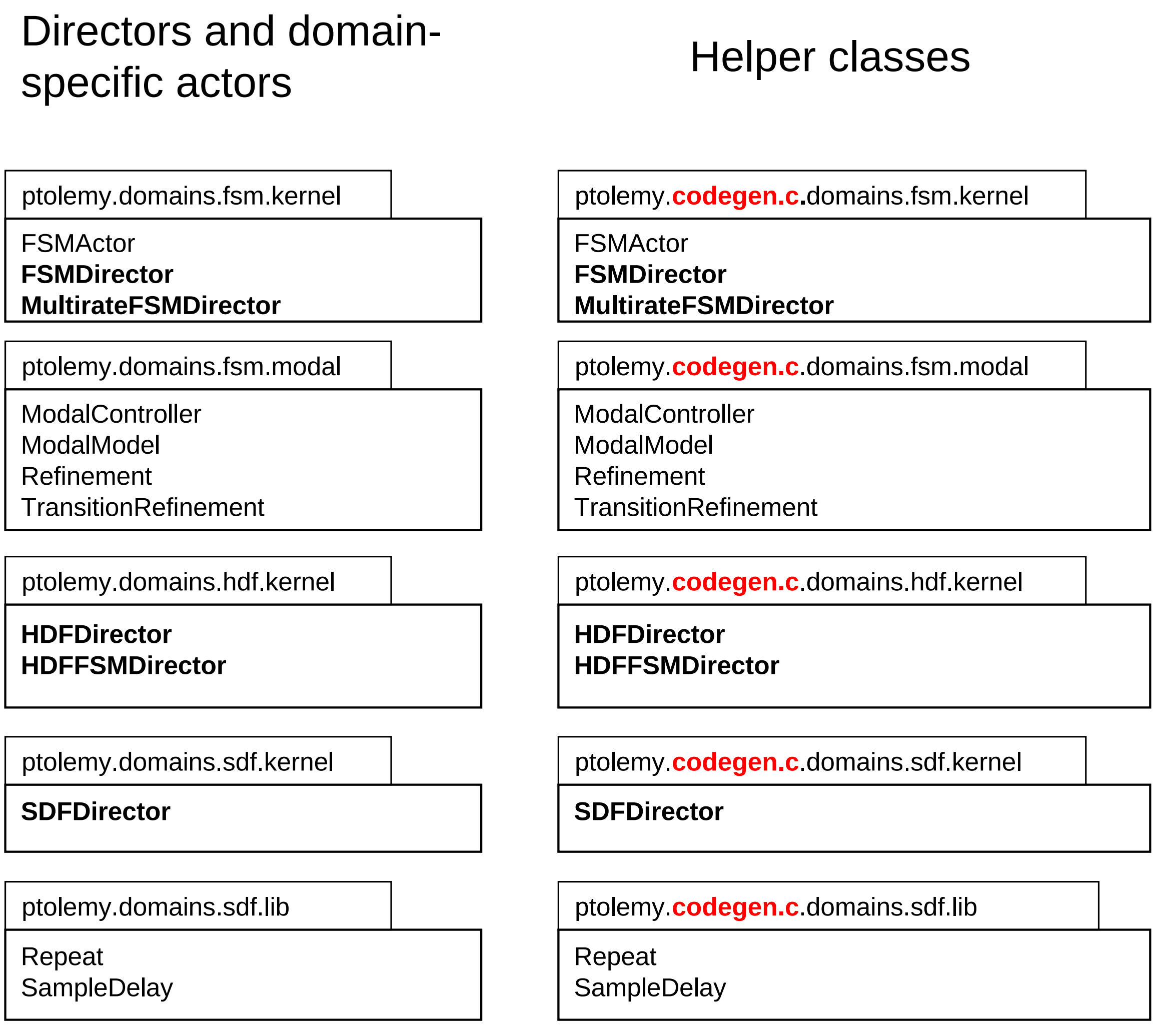


Figure 5.6: Directors, domain-specific actors and corresponding helpers.

parameters need to be defined as variables. However, whether a parameter is static or dynamic can only be determined through model analysis after processing all the code blocks. Therefore the variable definitions for the dynamic parameters are generated last and then placed at the beginning of the code.

The helper based code generation framework actually serves as a coordination language for the target code. It leverages the huge legacy code repository. This is especially relevant for embedded system design since it can easily incorporate pre-existing wrapper code around hardware. It takes advantage of many years and many researchers' work on compiler optimization techniques for the target language, such as C. My partial evaluation techniques are mostly applied at the component level. As pointed out by [Schultz2003], compilers apply a wider range of optimizations, at the language level, such as copy propagation and loop invariant removal that do not necessarily depend on statically determined constants. Partial evaluation is thus dependent on a compiler for traditional intra-procedural optimizations such as copy propagation, common subexpression elimination, loop invariant removal, etc. that are essential for good performance. Furthermore, optimizations that are not expressible at the language level, such as register allocation and array bounds check elimination, cannot be performed by a partial evaluator, and must be handled by a compiler.

My code generation framework also has the advantage of being accessible to a huge base of programmers. Often new languages fail to catch on not because it is technically flawed, but because it is very difficult to penetrate the barrier established

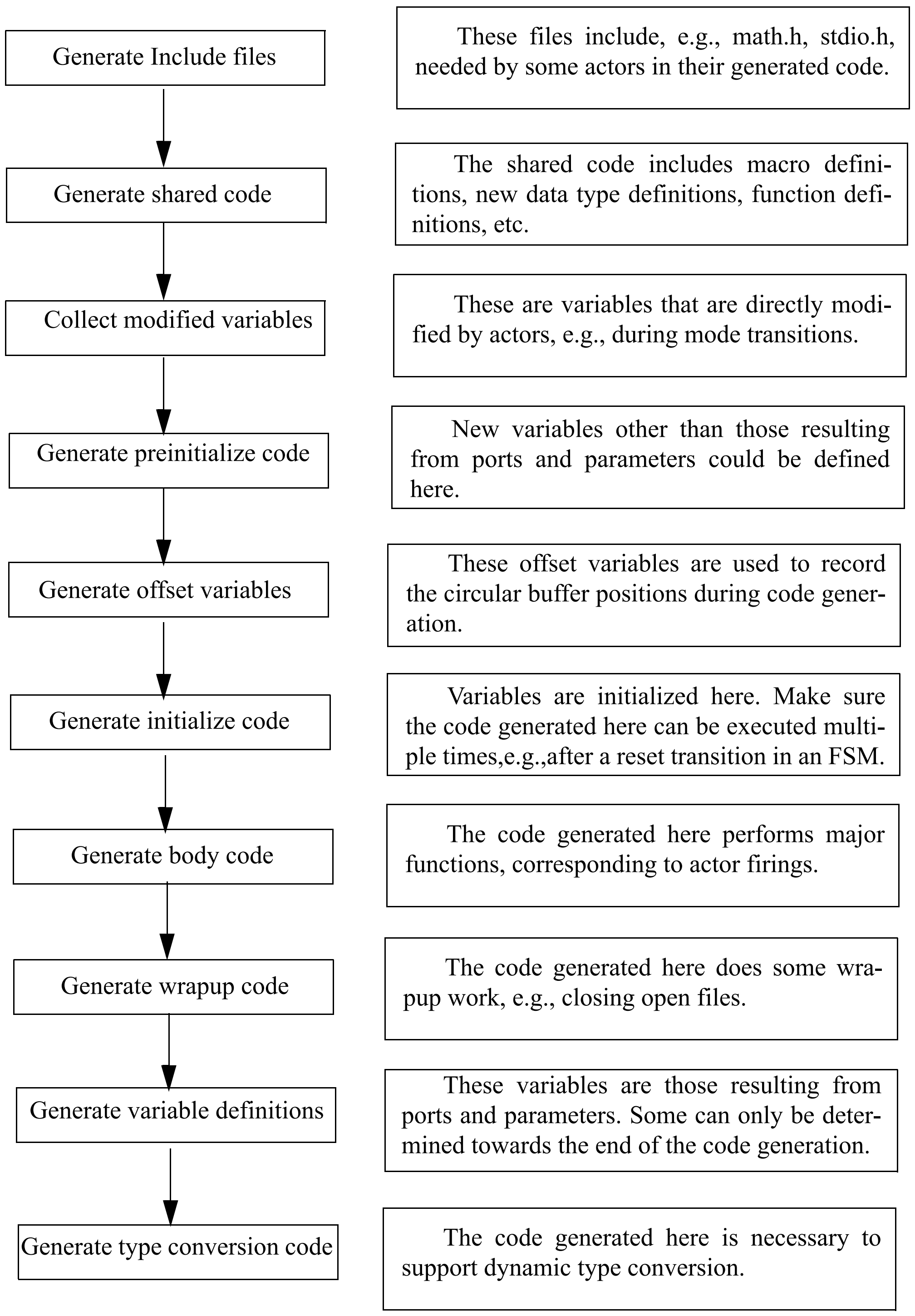

Generate Include files
These files include, e.g., math.h, stdio.h, needed by some actors in their generated code.
Generate shared code
The shared code includes macro definitions, new data type definitions, function definitions, etc.
Collect modified variables
These are variables that are directly modified by actors, e.g., during mode transitions.
Generate preinitialize code
New variables other than those resulting from ports and parameters could be defined here.
Generate offset variables
These offset variables are used to record the circular buffer positions during code generation.
Generate initialize code
Variables are initialized here. Make sure the code generated here can be executed multiple times,e.g.,after a reset transition in an FSM.
Generate body code
The code generated here performs major functions, corresponding to actor firings.
Generate wrapup code
The code generated here does some wrapup work, e.g., closing open files.
Generate variable definitions
These variables are those resulting from ports and parameters. Some can only be determined towards the end of the code generation.
Generate type conversion code
The code generated here is necessary to support dynamic type conversion.

Figure 5.7: The flow chart of the code generation process.

by the languages already in widespread use. With the use of the helper class combined with target code template written in a language programmers are familiar with, there is much less of a learning curve to use such an environment.

5.5 Domains

5.5.1 SDF

The synchronous dataflow (SDF) domain [Lee1987] is one of the most mature domains in Ptolemy II. Under the SDF domain, the execution order of actors is statically determined prior to execution. This opens the door for generating some very efficient code. In fact, the SDF software synthesis has been studied extensively. One book [Bhattacharyya1996], several Ph.D. dissertations [Bhattacharyya1994][Murthy1996] and numerous papers have been written. Many optimization techniques have been designed according to different criteria such as minimization of program size, buffer size, or actor activation rate. Hardware synthesis from SDF specification has also been studied by many researchers, e.g., see [Horstmannshoff2000]. I built the support for SDF code generation to test my framework and use it as a starting point to explore code generation for other domains.

The helper for the SDFDirector allows to plug in any optimizing SDFScheduler. The maximum size of each buffer is determined by the SDFScheduler and the helper uses that to determine the array size in the generated code (for buffer size 1, only a simple variable instead of an array is defined). There are two modes for the generated code. In the inline mode, actor firing code is stacked on top of each other. In the non-inline mode, the firing code of each actor is wrapped in its own individual function. The inline version may run faster without call-return overhead. The non-inline version may reduce memory footage when there is no single appearance schedule [Bhattacharyya1994][Murthy1996] or when the reduction of buffer size offered by multiple appearance schedule is more effective than the reduction of program size using single appearance schedule.

5.5.2 FSM

Finite state machines (FSM's) have been the subject of a long history of research work. There has been more recent work on hierarchical concurrent finite state machines [Girault1999]. In Ptolemy II, an FSM actor serves two purposes: for traditional FSM modeling or for building modal models. In traditional FSM modeling, an FSM actor reacts to the inputs by making state transitions and sending data to the output ports like a regular Ptolemy actor.

The FSM domain also supports the hierarchical concurrent finite state machines with modal models. In Fig. 5.8, M is a modal model with two operation modes. Modes are represented by states (rendered as circles in the figure) of an FSM actor that controls mode switching. Each mode has one or more refinements that specify the behavior of the mode. A modal model is constructed in a ModalModel actor

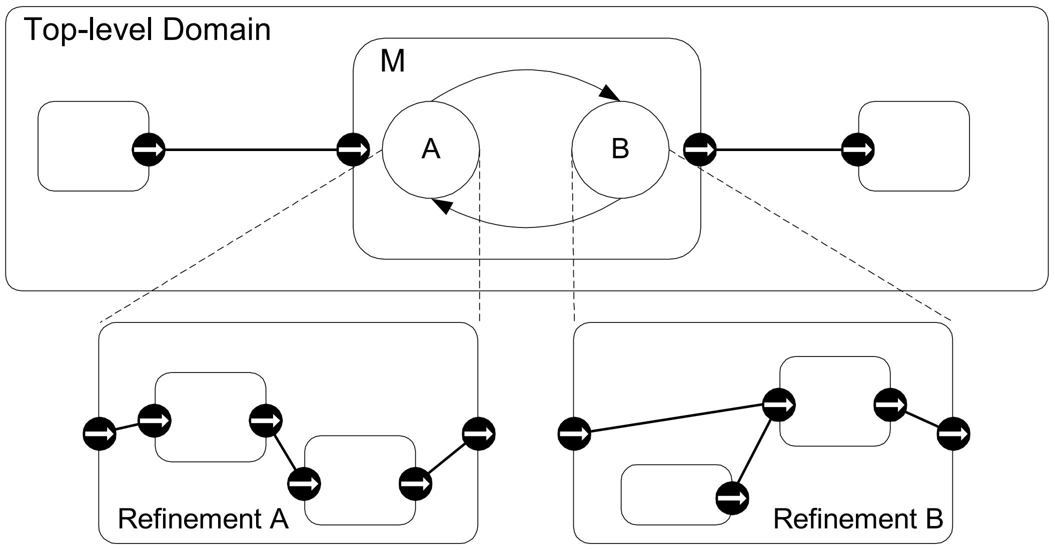


Figure 5.8: A modal model example.

having the *FSMDirector as the local director. The ModalModel actor contains a ModalController (essentially an FSM actor) and a set of Refinement actors that model the refinements associated with states and possibly a set of TransitionRefinement actors that model the refinements associated with transitions.

The *FSMDirector (which could be FSMDirector, MultirateFSMDirector, or HDFF-SMDirector) mediates the interaction with the outside domain, and coordinates the execution of the refinements with the ModalController. A modal model controlled by FSMDirector can only consume and produce at most one token at each port. A MultirateFSMDirector supports multi-token syntax and therefore can control multi-rate modal models. Both FSMDirector and MultirateFSMDirector try to make a state transition after each firing of the modal model, which results in Turing-completeness. Therefore a model designed using these two directors are generally not statically schedulable. However, as discussed in Section 3.4, if a modal model has the same rate signature at different states, it could not only generate static schedules but also reduce the scheduling complexity. If a modal model has different rate signatures at different states, we need to constrait state transitions to happen only between global iterations to have a statically schedulable model and HDFFSMDirector does exactly that.

In order to generate code for a modal model, I created a helper class for FSMActor which can take both roles of a standalone actor and a modal controller.[3] I designed the helper so that it can generate appropriate code according to its context. Specifically, it implements a method

```
generateTransitionCode(StringBuffer, TransitionRetriever)
```

where TransitionRetriever is a Java interface with a method

```
Iterator retrieveTransitions(State).
```

When the FSMActor is used as a standalone actor, the TransitionRetriever argument is given an instance of an anonymous class that implements the retrieveTransitions(State) method in such a way that it returns an Iterator of all transitions from

[3]For the use and implementation of the FSMActor itself, refer to Chapter 4 of [Brooks2007-3].

the given State; when the FSMActor is used as a modal controller, the Transition-Retriever argument is given an instance of an anonymous class that implements the retrieveTransitions(State) method in such a way that it returns an Iterator of either preemptive or non-preemptive transitions from the given State.[4] The code generated in the generateTransitionCode(StringBuffer,TransitionRetriever) method follows exactly the same execution sequence as in the simulation model. For each state and the transitions from that state, it generates the code for guard expression, choice action (which can only produce output and cannot change actor state), transition refinement, commit action (which can do both), new state updating and re-initialization (if required).

5.5.3 HDF

An HDF model allows changes in rate signatures between iterations of the whole model. Within each iteration, rate signatures are fixed and an HDF model behaves like an SDF model. This guarantees that a schedule can be executed to the completion. Between global iterations, a modal model can make a state transition and derives its new rate signature from the refinement associated with the new state. The HDF domain recomputes the schedule as necessary.

The HDF domain can be used to model a variety of interesting applications that

[4]A modal controller distinguishes between preemptive transitions (which can be taken before executing the refinement for the current state) and non-preemptive transitions (which can only be taken after executing the refinement for the current state) while a standalone FSMActor does not have such concept.

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

Since it's expensive to compute the schedule during the run time, all possible schedules are precomputed during code generation. The structure of the generated code is hard-coded in such a way that it reflects all possible execution paths for different schedules.

The first step in the code generation for HDF models is to do configuration analysis as presented in section 3.3 in a *bottom-up* fashion. When the analysis is complete, each actor has recorded the number of internal configurations, the rate signature presented to the outside in each configuration, and the SDF schedule in each configuration. During this step, the maximum capacity required of each buffer among all configurations is also recorded, except the input buffers of the modal controller. Remember that the modal controller processes all received data only after one global iteration and determines whether it needs to make a transition at that point. Therefore it needs to buffer all received data from one global iteration. However, how many data tokens are received in one global iteration depends on how many times the modal model containing the controller is fired. That information is not available yet in the *bottom-up* traversal since it depends on the schedules higher up in the hierarchy.

The next step is to traverse the model structure in a *top-down* fashion. Each composite actor is associated with an integer array `_firingsPerGlobalIteration`. The length of the array equals the number of internal configurations of that actor. Each element in the array represents the maximum number of times the actor can be fired in one global iteration when the actor is in the corresponding internal configuration. It is computed the following way. If a composite actor is internally controlled by an HDFDirector, each contained actor derives its `_firingsPerGlobalIteration` from that of its container and the number of times this actor is fired in a local SDF schedule. If a composite actor is internally controlled by an HDFFSMDirector (i.e., it is a ModalModel), each contained refinement derives its `_firingsPerGlobalIteration` directly from that of its container. The number of tokens a modal controller receives in one global iteration could be potentially large, implying a large chunk of memory must be allocated in the generated code. One way to optimize this is to directly analyze the guard expression and find out how far back the controller needs to access the received tokens and only allocate the necessary amount of memory. This should be easy if *constant* array indexes are used to access the received tokens, which is the normal usage. However, if array indexes are variables, the analysis would be difficult or even impossible. The current implementation allocates the maximum amount of memory needed for one global iteration, therefore correctness is guaranteed.

I'll use the HDF model in Fig. 5.9 to explain many aspects of the partial evaluation techniques used in the code generation framework. The HDF model contains a modal model with two states. The refinement associated with state1 has a rate signature of {1,1} and the refinement associated with state2 has a rate signature of {1, repeatFactor} where repeatFactor is a parameter that can be configured by users. The modal model implicitly contains an HDFFSMDirector.

Ptolemy II actors are type-polymorphic, which means the same actor can operate on different data types—not only scalar data types but also structured data types. This capability comes at the cost of performance—a simple addition can involve about a dozen function calls. The partial evaluation technique transforms a type-polymorphic specification to a specification that only operates on a single type. In Fig. 5.9, the parameters of the Ramp actor are configured as double data type. The type-resolution mechanism could propagate this information throughout the model. In the generated code, all the variables are defined with specific data types (see Fig. 5.10) and all the operations are specific to these data types.

In the simulation environment memory can be dynamically allocated since the underlying framework must support those concurrency formalisms that are statically schedulable as well as those that are not. In code generation if a model is statically schedulable, the scheduling analysis could determine the buffer size and the generated code could pre-allocate the required memory. In Fig. 5.9, the value of the parameter `repeatFactor` in fact determines the size of the buffer receiving data from Repeat

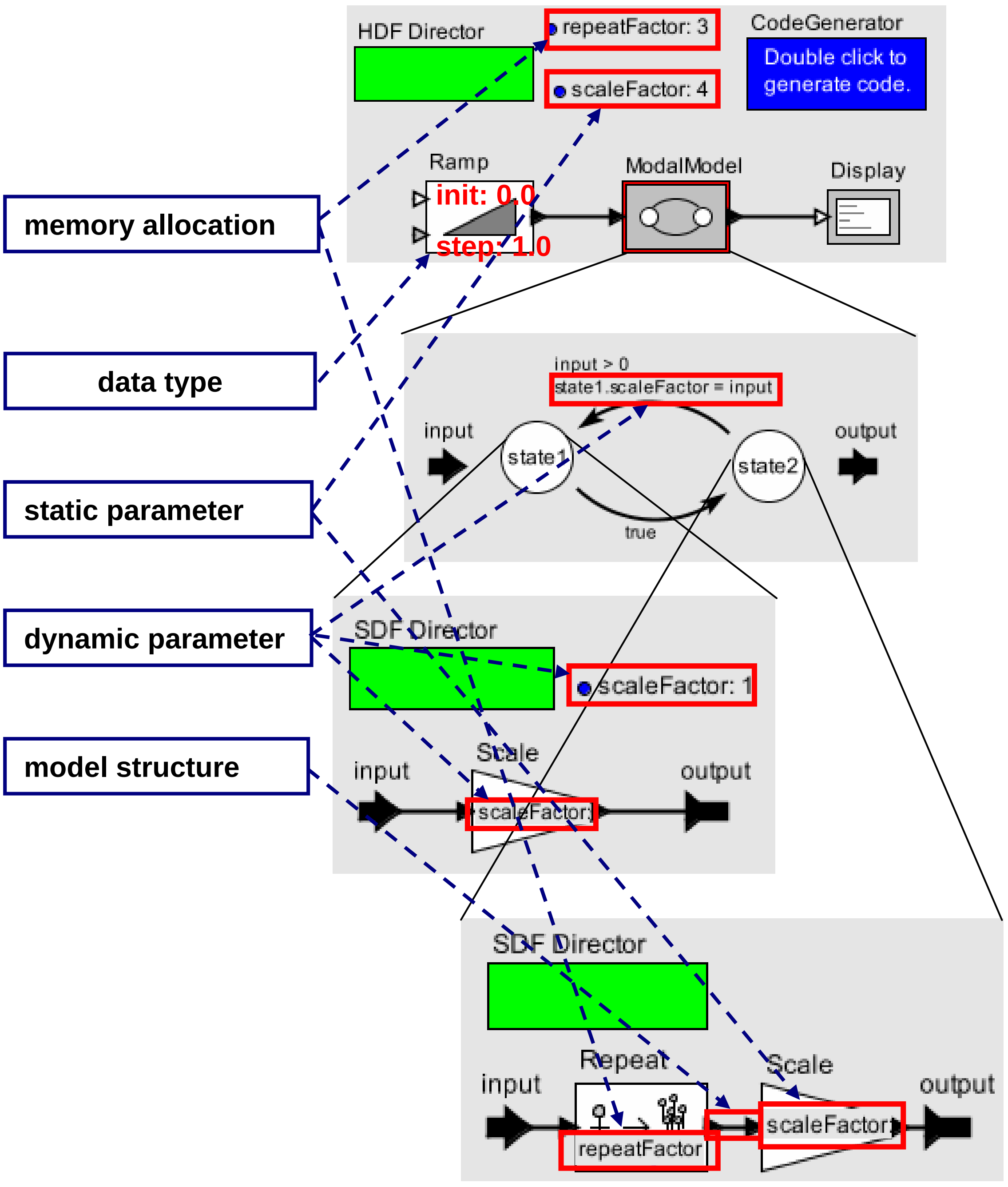


Figure 5.9: An HDF model used for code generation.

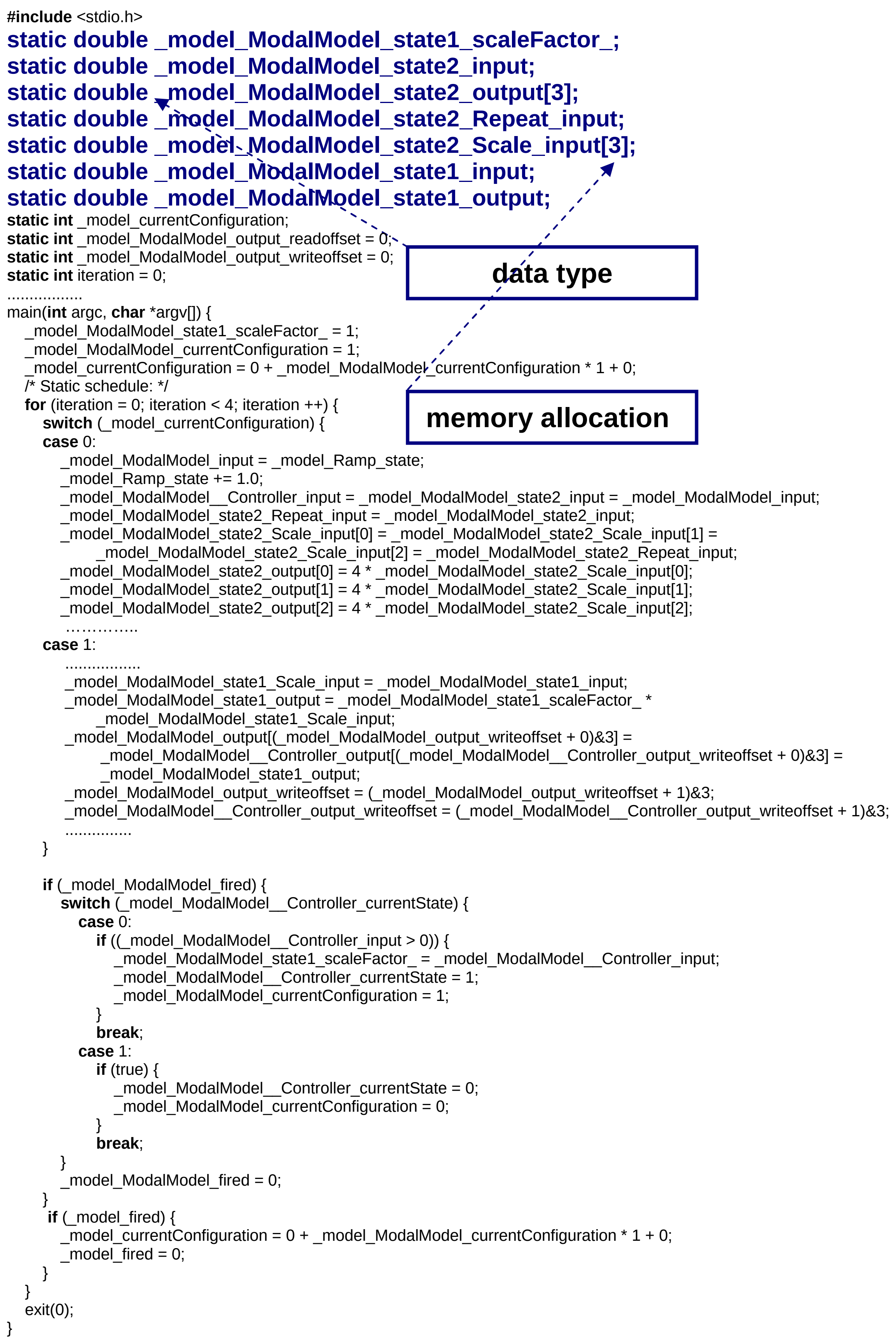
actor. In the generated code, the buffers are defined in the form of arrays of fixed sizes or scalar variables if size is 1 (see Fig. 5.10).

Through the model analysis we could divide parameters into static and dynamic. In Fig. 5.9, the `scaleFactor` parameter is defined in the top level actor and used in the Scale actor contained in the refinement for state2. This parameter can only be configured by users and stay constant during execution. Therefore it is static and directly used in the generated code (see Fig. 5.11). On the other hand, the `scaleFactor` parameter defined inside the refinement for state1 is dynamic because it can be modified each time there's a transition from state2 to state1.[5] A variable needs to be generated for the dynamic parameter (see Fig. 5.12).

Ptolemy II actors have well-defined interfaces, which provide abstraction for the properties of actors. Different actors can work together as long as the interfaces are compatible. When passing tokens between actors, an actor does not directly reference the downstream actor. Instead, the message passing is through the interface methods. However, most models have static structures. The partial evaluation technique takes advantage of this and replaces interface methods with direct reference to the buffer variables (see Fig. 5.11).

Ptolemy II actors are designed to be domain-polymorphic, meaning they can be used in different MoC's. The partial evaluation technique takes advantage of the specific MoC and reduces the generated code for the actor to the bare minimum

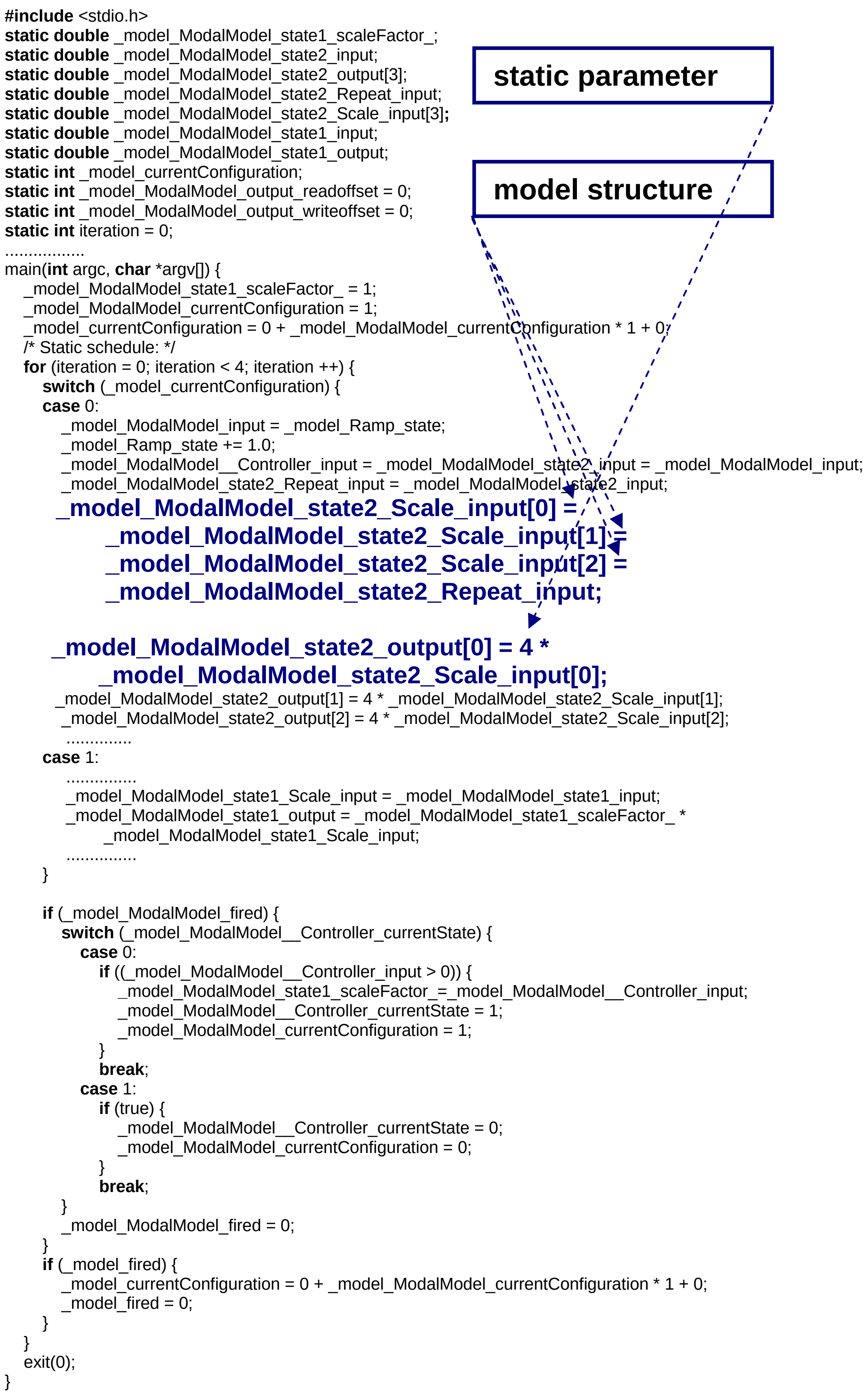
[5]Like scoping rule in general purpose languages such as C and Java, the inner parameter shadows the outer parameter in Ptolemy II. When an actor uses a parameter, it searches up in the hierarchy and uses the first matched parameter.



```
#include <stdio.h>
static double _model_ModalModel_state1_scaleFactor_;
static double _model_ModalModel_state2_input;
static double _model_ModalModel_state2_output[3];
static double _model_ModalModel_state2_Repeat_input;
static double _model_ModalModel_state2_Scale_input[3];
static double _model_ModalModel_state1_input;
static double _model_ModalModel_state1_output;
static int _model_currentConfiguration;
static int _model_ModalModel_output_readoffset = 0;
static int _model_ModalModel_output_writeoffset = 0;
static int iteration = 0;
.................
main(int argc, char *argv[]) {
  _model_ModalModel_state1_scaleFactor_ = 1;
  _model_ModalModel_currentConfiguration = 1;
  _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
  /* Static schedule: */
  for (iteration = 0; iteration < 4; iteration ++) {
    switch (_model_currentConfiguration) {
    case 0:
      _model_ModalModel_input = _model_Ramp_state;
      _model_Ramp_state += 1.0;
      _model_ModalModel__Controller_input = _model_ModalModel_state2_input = _model_ModalModel_input;
      _model_ModalModel_state2_Repeat_input = _model_ModalModel_state2_input;
      _model_ModalModel_state2_Scale_input[0] = _model_ModalModel_state2_Scale_input[1] =
          _model_ModalModel_state2_Scale_input[2] = _model_ModalModel_state2_Repeat_input;
      _model_ModalModel_state2_output[0] = 4 * _model_ModalModel_state2_Scale_input[0];
      _model_ModalModel_state2_output[1] = 4 * _model_ModalModel_state2_Scale_input[1];
      _model_ModalModel_state2_output[2] = 4 * _model_ModalModel_state2_Scale_input[2];
       ……………
    case 1:
       .................
       _model_ModalModel_state1_Scale_input = _model_ModalModel_state1_input;
       _model_ModalModel_state1_output = _model_ModalModel_state1_scaleFactor_ *
           _model_ModalModel_state1_Scale_input;
       _model_ModalModel_output[(_model_ModalModel_output_writeoffset + 0)&3] =
           _model_ModalModel__Controller_output[(_model_ModalModel__Controller_output_writeoffset + 0)&3] =
           _model_ModalModel_state1_output;
       _model_ModalModel_output_writeoffset = (_model_ModalModel_output_writeoffset + 1)&3;
       _model_ModalModel__Controller_output_writeoffset = (_model_ModalModel__Controller_output_writeoffset + 1)&3;
       ...............
    }

    if (_model_ModalModel_fired) {
      switch (_model_ModalModel__Controller_currentState) {
        case 0:
          if ((_model_ModalModel__Controller_input > 0)) {
            _model_ModalModel_state1_scaleFactor_ = _model_ModalModel__Controller_input;
            _model_ModalModel__Controller_currentState = 1;
            _model_ModalModel_currentConfiguration = 1;
          }
          break;
        case 1:
          if (true) {
            _model_ModalModel__Controller_currentState = 0;
            _model_ModalModel_currentConfiguration = 0;
          }
          break;
      }
      _model_ModalModel_fired = 0;
    }
     if (_model_fired) {
      _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
      _model_fired = 0;
    }
  }
  exit(0);
}
```

Figure 5.10: Data type and memory allocation in code generation.

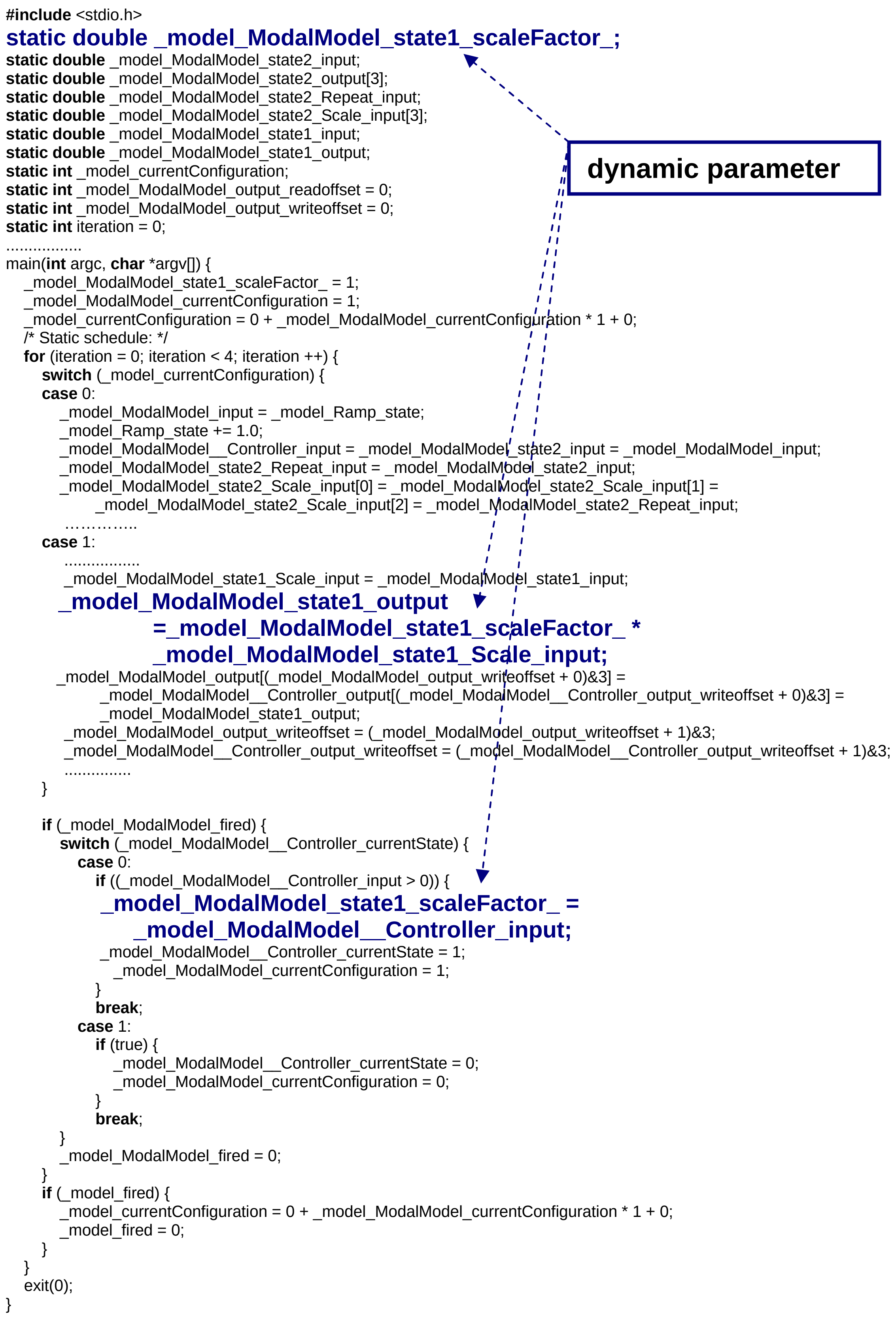


```
#include <stdio.h>
static double _model_ModalModel_state1_scaleFactor_;
static double _model_ModalModel_state2_input;
static double _model_ModalModel_state2_output[3];
static double _model_ModalModel_state2_Repeat_input;
static double _model_ModalModel_state2_Scale_input[3];
static double _model_ModalModel_state1_input;
static double _model_ModalModel_state1_output;
static int _model_currentConfiguration;
static int _model_ModalModel_output_readoffset = 0;
static int _model_ModalModel_output_writeoffset = 0;
static int iteration = 0;
.................
main(int argc, char *argv[]) {
   _model_ModalModel_state1_scaleFactor_ = 1;
   _model_ModalModel_currentConfiguration = 1;
   _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
   /* Static schedule: */
   for (iteration = 0; iteration < 4; iteration ++) {
      switch (_model_currentConfiguration) {
      case 0:
         _model_ModalModel_input = _model_Ramp_state;
         _model_Ramp_state += 1.0;
         _model_ModalModel__Controller_input = _model_ModalModel_state2_input = _model_ModalModel_input;
         _model_ModalModel_state2_Repeat_input = _model_ModalModel_state2_input;
        _model_ModalModel_state2_Scale_input[0] =
                _model_ModalModel_state2_Scale_input[1] =
                _model_ModalModel_state2_Scale_input[2] =
                _model_ModalModel_state2_Repeat_input;

       _model_ModalModel_state2_output[0] = 4 *
               _model_ModalModel_state2_Scale_input[0];
        _model_ModalModel_state2_output[1] = 4 * _model_ModalModel_state2_Scale_input[1];
         _model_ModalModel_state2_output[2] = 4 * _model_ModalModel_state2_Scale_input[2];
          ..............
      case 1:
          ...............
          _model_ModalModel_state1_Scale_input = _model_ModalModel_state1_input;
          _model_ModalModel_state1_output = _model_ModalModel_state1_scaleFactor_ *
                _model_ModalModel_state1_Scale_input;
          ...............
      }

      if (_model_ModalModel_fired) {
         switch (_model_ModalModel__Controller_currentState) {
            case 0:
               if ((_model_ModalModel__Controller_input > 0)) {
                  _model_ModalModel_state1_scaleFactor_=_model_ModalModel__Controller_input;
                  _model_ModalModel__Controller_currentState = 1;
                  _model_ModalModel_currentConfiguration = 1;
               }
               break;
            case 1:
               if (true) {
                  _model_ModalModel__Controller_currentState = 0;
                  _model_ModalModel_currentConfiguration = 0;
               }
               break;
         }
         _model_ModalModel_fired = 0;
      }
      if (_model_fired) {
         _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
         _model_fired = 0;
      }
   }
   exit(0);
}
```

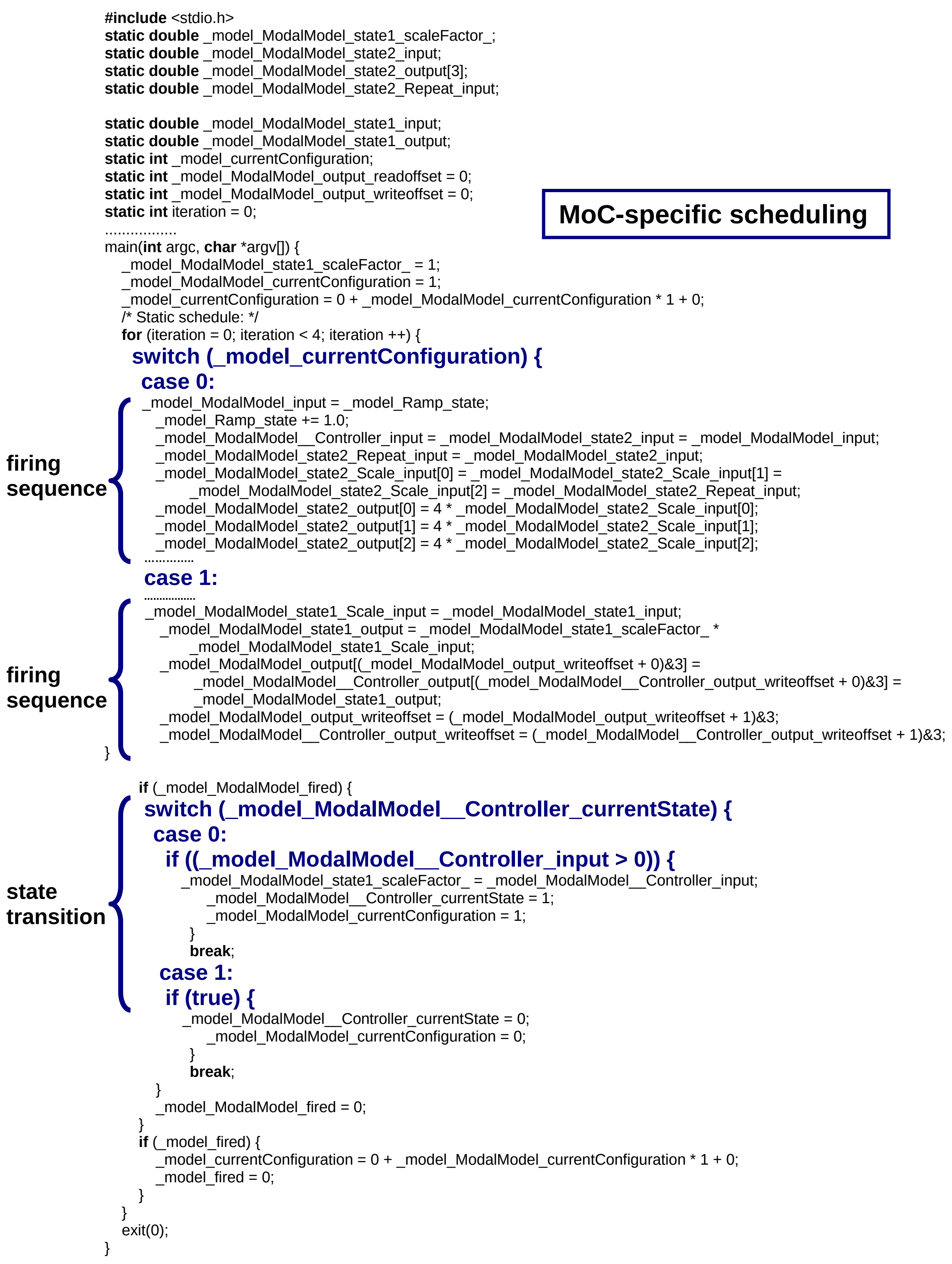
Figure 5.11: Static parameter and model structure in code generation.



```
#include <stdio.h>
static double _model_ModalModel_state1_scaleFactor_;
static double _model_ModalModel_state2_input;
static double _model_ModalModel_state2_output[3];
static double _model_ModalModel_state2_Repeat_input;
static double _model_ModalModel_state2_Scale_input[3];
static double _model_ModalModel_state1_input;
static double _model_ModalModel_state1_output;
static int _model_currentConfiguration;
static int _model_ModalModel_output_readoffset = 0;
static int _model_ModalModel_output_writeoffset = 0;
static int iteration = 0;
.................
main(int argc, char *argv[]) {
  _model_ModalModel_state1_scaleFactor_ = 1;
  _model_ModalModel_currentConfiguration = 1;
  _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
  /* Static schedule: */
  for (iteration = 0; iteration < 4; iteration ++) {
    switch (_model_currentConfiguration) {
    case 0:
      _model_ModalModel_input = _model_Ramp_state;
      _model_Ramp_state += 1.0;
      _model_ModalModel__Controller_input = _model_ModalModel_state2_input = _model_ModalModel_input;
      _model_ModalModel_state2_Repeat_input = _model_ModalModel_state2_input;
      _model_ModalModel_state2_Scale_input[0] = _model_ModalModel_state2_Scale_input[1] =
              _model_ModalModel_state2_Scale_input[2] = _model_ModalModel_state2_Repeat_input;
       …………..
    case 1:
       .................
       _model_ModalModel_state1_Scale_input = _model_ModalModel_state1_input;
      _model_ModalModel_state1_output
                    =_model_ModalModel_state1_scaleFactor_ *
                    _model_ModalModel_state1_Scale_input;
      _model_ModalModel_output[(_model_ModalModel_output_writeoffset + 0)&3] =
              _model_ModalModel__Controller_output[(_model_ModalModel__Controller_output_writeoffset + 0)&3] =
              _model_ModalModel_state1_output;
       _model_ModalModel_output_writeoffset = (_model_ModalModel_output_writeoffset + 1)&3;
       _model_ModalModel__Controller_output_writeoffset = (_model_ModalModel__Controller_output_writeoffset + 1)&3;
       ...............
    }

    if (_model_ModalModel_fired) {
      switch (_model_ModalModel__Controller_currentState) {
        case 0:
          if ((_model_ModalModel__Controller_input > 0)) {
           _model_ModalModel_state1_scaleFactor_ =
               _model_ModalModel__Controller_input;
           _model_ModalModel__Controller_currentState = 1;
             _model_ModalModel_currentConfiguration = 1;
          }
          break;
        case 1:
          if (true) {
            _model_ModalModel__Controller_currentState = 0;
            _model_ModalModel_currentConfiguration = 0;
          }
          break;
      }
      _model_ModalModel_fired = 0;
    }
    if (_model_fired) {
      _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
      _model_fired = 0;
    }
  }
  exit(0);
}
```

Figure 5.12: Dynamic parameter in code generation.


MoC-specific scheduling
firing sequence
firing sequence
state transition

```
#include <stdio.h>
static double _model_ModalModel_state1_scaleFactor_;
static double _model_ModalModel_state2_input;
static double _model_ModalModel_state2_output[3];
static double _model_ModalModel_state2_Repeat_input;

static double _model_ModalModel_state1_input;
static double _model_ModalModel_state1_output;
static int _model_currentConfiguration;
static int _model_ModalModel_output_readoffset = 0;
static int _model_ModalModel_output_writeoffset = 0;
static int iteration = 0;
................
main(int argc, char *argv[]) {
   _model_ModalModel_state1_scaleFactor_ = 1;
   _model_ModalModel_currentConfiguration = 1;
   _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
   /* Static schedule: */
   for (iteration = 0; iteration < 4; iteration ++) {
     switch (_model_currentConfiguration) {
      case 0:
      _model_ModalModel_input = _model_Ramp_state;
         _model_Ramp_state += 1.0;
         _model_ModalModel__Controller_input = _model_ModalModel_state2_input = _model_ModalModel_input;
         _model_ModalModel_state2_Repeat_input = _model_ModalModel_state2_input;
         _model_ModalModel_state2_Scale_input[0] = _model_ModalModel_state2_Scale_input[1] =
               _model_ModalModel_state2_Scale_input[2] = _model_ModalModel_state2_Repeat_input;
         _model_ModalModel_state2_output[0] = 4 * _model_ModalModel_state2_Scale_input[0];
         _model_ModalModel_state2_output[1] = 4 * _model_ModalModel_state2_Scale_input[1];
         _model_ModalModel_state2_output[2] = 4 * _model_ModalModel_state2_Scale_input[2];
      .............
      case 1:
      ...............
      _model_ModalModel_state1_Scale_input = _model_ModalModel_state1_input;
         _model_ModalModel_state1_output = _model_ModalModel_state1_scaleFactor_ *
               _model_ModalModel_state1_Scale_input;
         _model_ModalModel_output[(_model_ModalModel_output_writeoffset + 0)&3] =
               _model_ModalModel__Controller_output[(_model_ModalModel__Controller_output_writeoffset + 0)&3] =
               _model_ModalModel_state1_output;
         _model_ModalModel_output_writeoffset = (_model_ModalModel_output_writeoffset + 1)&3;
         _model_ModalModel__Controller_output_writeoffset = (_model_ModalModel__Controller_output_writeoffset + 1)&3;
}

      if (_model_ModalModel_fired) {
       switch (_model_ModalModel__Controller_currentState) {
        case 0:
         if ((_model_ModalModel__Controller_input > 0)) {
          _model_ModalModel_state1_scaleFactor_ = _model_ModalModel__Controller_input;
              _model_ModalModel__Controller_currentState = 1;
              _model_ModalModel_currentConfiguration = 1;
            }
            break;
        case 1:
         if (true) {
          _model_ModalModel__Controller_currentState = 0;
              _model_ModalModel_currentConfiguration = 0;
            }
            break;
         }
         _model_ModalModel_fired = 0;
      }
      if (_model_fired) {
         _model_currentConfiguration = 0 + _model_ModalModel_currentConfiguration * 1 + 0;
         _model_fired = 0;
      }
   }
   exit(0);
}
```

Figure 5.13: MoC-specific scheduling in code generation.

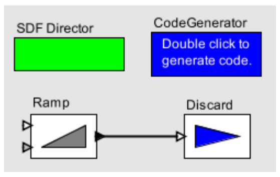
needed for that MoC. The partial evaluation technique also takes advantage of MoC-specific scheduling and the schedules are hard-coded directly into the flow of the generated code (see Fig. 5.13).

5.6 Performance

I compared the execution time of Ptolemy simulation and the corresponding generated C code. In Fig. 5.14, the first model is a simple producer and consumer pair; the second model is the one discussed in the previous section; the third one models a communication system using Hamming coder and decoder over a noisy channel.[6]

In each case, I recorded the number of iterations and the corresponding running time. Since C code execution is much faster than Java simulation, I adjusted the iteration number in each case so that the running time is always in the scale of 1 second. Then for each model, I computed the the ratio of model simulation time over C code execution time per iteration. As one can see from the table, the generated C code executes three orders of magnitude faster than the simulation in each case. Admittedly, part of performance difference comes from using partial evaluation techniques and part of performance difference comes from using two different languages—C being natively executed as machine code and Java being interpreted by JVM as bytecode. However, nowadays JVM implements just-in-time (JIT) technology which compiles the bytecode into platform-specific executable code. In the experiment I measured

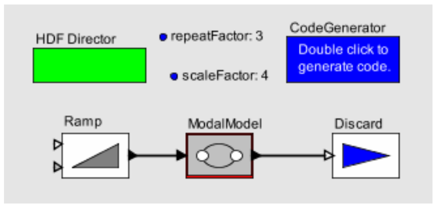
[6]This model is created by Ye Zhou.



	Number of iterations	Time(s)
Ptolemy II	10^5	0.33
C	10^8	0.34

Speedup

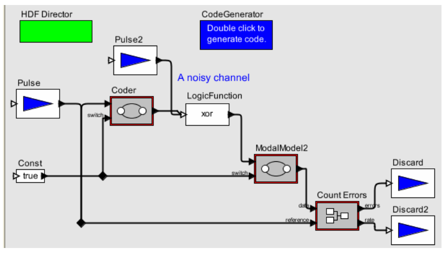
970



	Number of iterations	Time(s)
Ptolemy II	10^3	0.23
C	10^6	0.44

Speedup

520



	Number of iterations	Time(s)
Ptolemy II	10^3	0.53
C	10^6	0.75

Speedup

710

Figure 5.14: Performance comparison of the original model and generated code.

the simulation time only after running the same simulation a number of times so that JIT can kick in. In the future, I hope to do further experiments to determine more precisely the performance improvement from using partial evaluation techniques.

Chapter 6

Conclusion

6.1 Summary of Results

The first part of this thesis studied using actor-oriented design as a high level abstraction for composing concurrent components. Under the umbrella of actor-oriented design, there are a rich variety of concurrency formalisms that span the whole spectrum from most expressive to most analyzable. I focused on one particular model of computation called heterochronous dataflow (HDF) which strikes a nice balance between expressiveness and analyzability. I studied the configurations in HDF modeling paving a way for code generation. I also studied the complexity issues in HDF modeling.

In the second part of this thesis, I confronted the abstraction penalty problem under the context of actor-oriented design. To generate an efficient implementation from a high level specification, I used partial evaluation as an optimized compilation technique for actor-oriented models. Compared with traditional compiler optimization, partial evaluation for embedded software works at the component level and heavily leverages domain-specific knowledge. Through model analysis—the counterpart of binding-time analysis for general purpose software—the partial evaluator I developed can discover the static parts in a model including data types, buffer sizes, parameter values, model structures and execution schedules, etc. and then exploit the known information to reach a very efficient implementation.

I used a helper-based mechanism, which leads to a flexible and extensible framework and achieves modularity, maintainability, portability and efficiency in code generation. I demonstrated that design using high level abstraction can be achieved without sacrificing performance.

The code generation framework is part of Ptolemy II release. It can be downloaded in open source from the Ptolemy project website at EECS, UC Berkeley.[1] The software release includes various demos highlighting the features of the code generation framework.

[1]Go to http://ptolemy.eecs.berkeley.edu/ or http://ptolemy.org/

6.2 Future Work

There remains much to be explored.[2] Ptolemy II is a graphical design environment and throughout the thesis I used graphical specifications for HDF models, which has the advantage of being intuitive about the design. However, the most obvious weakness of graphical design is the issue of scalability. A textual specification handles better a large design. Therefore the open question is how to design a language with HDF semantics. One starting point is to adapt Cal [Eker2003], which is an actor definition language. The research challenge is to come up with a language that is built on sound design principles and leverages technology the Ptolemy project has made mature or has been experimenting with over the years, such as functional array operations, higher-order components, state machines, static schedulers, synchronization optimizers, etc.

This thesis did not touch upon the subject of parallel scheduling. In Ptolemy classic—the predecessor to Ptolemy II—a number of parallel scheduling algorithms for SDF models were realized. The research opportunity here is to build upon those results and take advantage of the functional expression language in Ptolemy II, which supports Matlab-like array operations, for exploiting the parallelism in HDF models.

In this thesis I have realized code generation for SDF, FSM and HDF domains. Since the beginning the framework is purposely designed to be flexible and extensible and allow code generation for multiple concurrency formalisms. Naturally the next

[2]Some ideas here come from Edward A. Lee's Parlab contributions email.

step is to explore code generation for other MoC's suited to embedded system design. E.g., Man-Kit Leung is building a C code generator for Process Network (PN) domain that will produce native multithreaded C code from Ptolemy II models.

Another direction to steer the code generation framework towards more practical use is to test the capabilities of the framework with more complicated applications. E.g., we could get some reference implementation for mpeg coding/decoding in C. We could do the same design in our actor-oriented environment and then automatically generate C code for that. Then we could compare the performance of generated code with that of the reference implementation.

Bibliography

[Simulink] http://www.mathworks.com/products/simulink/

[WDF] http://www.microsoft.com/whdc/driver/wdf/WDF_facts.mspx

[Sun] http://java.sun.com/javase/technologies/realtime/index.jsp

[Timesys] http://www.timesys.com/java/

[Agha1986] G. A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.

[Aho1986] A. V. Aho, R. Sethi and J. D. Ullman. Compilers: Principles, Techniques, and Tools. MA: Addison-Wesley, 1986.

[Alur1995] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Comput. Sci.*, vol. 138, no. 1, pp. 334, Feb. 1995.

[Anderson1992] L. O. Andersen. Partial evaluation of C and automatic compiler generation. In *4th International Conference CC'92 Proceedings.* Springer-Verlag. 1992, pp. 251-7.

[Anderson1994] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD. Thesis, DIKU, University of Copenhagen, May 1994.

[Bacon2005] D. F. Bacon, P. Cheng, D. Grove, M. Hind, V. T. Rajan, E. Yahav, M. Hauswirth, C. Kirsch, D. Spoonhower and M. T. Vechev. High-level Real-time Programming in Java. In *Proceedings of the Fifth ACM International Conference on Embedded Software*, Jersey City, New Jersey, September 2005.

[Benveniste1991] A. Benveniste and G. Berry. The synchronous approach to reactive real-time systems. In *Proceedings of the IEEE*, vol. 79, pp. 12701282, Sept. 1991.

[Benveniste2003] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone. The Synchronous Languages 12 Years Later. In *Proceedings of the IEEE*, vol. 91, pp. 64-83, Jan. 2003.

[Berry1989] G. Berry. Real Time programming: Special purpose or general purpose languages. In *Information Processing*, Ed. G. Ritter, Elsevier Science Publishers B.V. (North Holland), vol. 89, pp. 11-17, 1989.

[Belina1991] F. Belina, D. Hogrefe, and A. Sarma. SDL with Applications from Protocol Specification. Hemel Hempstead, U.K.: Prentice-Hall International, 1991.

[Bhattacharyya1994] S. S. Bhattacharyya. Compiling Dataflow Programs for Digital Signal Processing. *Tech. Report UCB/ERL 94/52*, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, July 12, 1994.

[Bhattacharyaand2000] B. Bhattacharyaand S. S. Bhattacharyya. Parameterized Dataflow Modeling of DSP Systems. In*International Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, June 2000.

[Bhattacharyya1996] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee. Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers, Norwell, Mass, 1996.

[Bondorf1990] A. Bondorf. Self-applicable partial evaluation. Ph.D. thesis, DIKU, University of Copenhagen, Denmark, 1990.

[Bondorf1991] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3-34, 1991.

[Bondorf1992] A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference in Lisp and Functional Programming*, San Francisco, California (Lisp Pointers, vol. V, no. 1, 1992), pp. 1-10, New York: ACM, 1992.

[Bondorf1991-2] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151-195, 1991.

[Boussinot1991] F. Boussinot and R. de Simone. The Esterel language. In *Proceedings of the IEEE*, vol. 79, pp. 12931304, Sept. 1991.

[Brooks2007-1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao and H. Zheng (eds.). Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). EECS Department, University of California, Berkeley, UCB/EECS-2007-7, January 11, 2007.

[Brooks2007-2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao and H. Zheng (eds.). Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture). EECS Department, University of California, Berkeley, UCB/EECS-2007-8, January 11, 2007.

[Brooks2007-3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao and H. Zheng (eds.). Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). EECS Department, University of California, Berkeley, UCB/EECS-2007-9, January 11, 2007.

[Buck1993] J. T. Buck. Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. *Technical Memorandum UCB/ERL 93/69*, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[Bulyonkov1988] M.A. Bulyonkov and A.P. Ershov. How Do Ad-Hoc Compiler Constructs Appear in Universal Mixed Computation Processes? *Partial Evaluation* *and Mixed Computation*, edited by D. Bjrner, A. P. Ershov and N.D. Jones, pp. 6581, North-Holland, 1988.

[Chiodo1994] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno and A. Sangiovanni-Vincentelli. Hardware-software codesign of embedded systems. *IEEE Micro*, pp. 2636, Aug. 1994.

[Eker2003] J. Eker and J. W. Janneck. CAL language report: Specification of the CAL actor language. *Technical Report Technical Memorandum No. UCB/ERL M03/48*, University of California, Berkeley, CA, December 1 2003.

[Engels1994] M. Engels, G. Bilsen, R. Lauwereins and J. Peperstraete. Cyclo-static data flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pp. 503-507, 1994.

[Ershov1977] A.P. Ershov. On the Partial Computation Principle. *Information Processing Letters* 6,2 (April 1977) 3841.

[Gamma1994] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Girault1999] A. Girault, B. Lee, and E. A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, Vol. 18, No. 6, June 1999.

[Gomard1991] C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, vol.1, no.1, Jan. 1991, pp. 21-69.

[Gomard1991-2] C.K. Gomard and N.D. Jones. Compiler Generation by Partial Evaluation: a Case Study. *Structured Programming*, 12 (1991), pp. 123-144.

[Halbwachs1993] N. Halbwachs. Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, 1993.

[Halbwachs1991] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, vol. 79, pp. 13051320, Sept. 1991.

[Hansen1988] C. Hansen. Hardware logic simulation by compilation. In *Proceedings of the Design Automation Conference (DAC)*, SIGDA, ACM, 1988.

[Harel1987] D. Harel. Statecharts: A visual formalism for complex systems. In *Sci. Comput. Program.*, vol. 8, pp. 231274, 1987.

[Henzinger2002] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN, ACM, June 2002.

[Hewitt1977] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323364, 1977.

[Horstmannshoff2000] J. Horstmannshoff and H. Meyr. Efficient Building Block Based RTL Code Generation from Synchronous Data Flow Graphs. In *Proceedings of the 37th conference on Design automation*, Los Angeles, California, United States, pp. 552 - 555, 2000.

[Jones1993] N. D. Jones, C. K. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall, June 1993.

[Kahn1974] G. Kahn .The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[Kleene1952] S.C. Kleene. Introduction to Metamathematics. Princeton, NJ: D. van Nostrand, 1952.

[Kohler2002] E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 251-263, October 2002.

[Bollella2000] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr and M. Turnbull. The Real-Time Specification for Java. Addison Wesley Longman, 2000.

[Le Guernic1991] P. Le Guernic, T. Gautier, M. Le Borgne and C. Le Maire. Programming real-time applications with SIGNAL. In *Proceedings of the IEEE*, vol. 79, pp. 13211336, Sept. 1991.

[Lee2006] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33-42, May 2006.

[Lee2008] E. A. Lee. Cyber Physical Systems: Design Challenges. EECS Department, University of California, Berkeley, *Technical Report No. UCB/EECS-2008-8*, January 23, 2008.

[Lee2005] E. A. Lee. Concurrent Models of Computation for Embedded Software. Lecture Notes for EECS 290N, Advanced Topics in Systems Theory, *Technical Memorandum UCB/ERL M05/2*, January 4, 2005, University of California, Berkeley, CA 94720.

[Lee1987] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. of the IEEE*, September, 1987.

[Lee1987-2] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, January, 1987.

[Lee1998] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. it IEEE Transactions on CAD, Vol. 17, No. 12, December 1998.

[Liao1997] S. Y. Liao, S. Tjiang and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th Design Automation Conference (DAC'1997)*, SIGDA, ACM, 1997.

[Lloyd1991] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, vol.11, no.3-4, Oct.-Nov. 1991, pp. 217-42.

[Maraninchi1991] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[Murthy1996] P. K. Murthy. Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow. *Technical Memorandum UCB/ERL M96/79*, Ph.D. Thesis, EECS Department, University of California, Berkeley, CA 94720, December 1996.

[Myers2004] G. J. Myers, C. Sandler, T. Badgett and T. M. Thomas. The Art of Software Testing. Second Edition, Wiley, 2004.

[Neuendorffer2004] S. Neuendorffer. Actor-Oriented Metaprogramming. Ph.D. Dissertation, *Technical Memorandum No. UCB/ERL M05/1*, University of California, Berkeley, December 21, 2004.

[Ommerling2000] R. V. Ommerling. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):7885, March 2000.

[Parks1995] T. M. Parks. Bounded Scheduling of Process Networks. *Technical Report*

UCB/ERL-95-105, PhD Dissertation, EECS Department, University of California, Berkeley, CA 94720, December 1995.

[Pino1995] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software Synthesis for DSP Using Ptolemy. *Journal on VLSI Signal Processing*, vol. 9, no. 1, pp. 7-21, Jan., 1995.

[Poore2004] J. Poore. A Tale of Three Disciplines ...and a Revolution. *IEEE Computer*, Vol. 37, No. 1, pp. 30-36, Jan., 2004.

[Schultz2001] U. P. Schultz. Partial evaluation for class-based object-oriented languages. In *Proceedings of Symposium on Programs as Data Objects (PADO)*, number 2053 in Lecture Notes in Computer Science. Springer-Verlag, May 2001.

[Schultz2000] U. P. Schultz. Object-Oriented Software Engineering using Partial Evaluation. PhD. Thesis, Universit de Rennes 1, December 2000.

[Schultz2003] U. P. Schultz, J. L. Lawall and C. Consel. Automatic Program Specialization for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 25(4), 2003.

[Sipser1996] M. Sipser. Introduction to the Theory of Computation. International Thomson Publishing, 1996.

[Teich1999] J. Teich, E. Zitzler and S. Bhattacharyya. 3D exploration of software schedules for DSP algorithms. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*, SIGDA, ACM, May 1999.

[Thies2002] W. Thies, M. Karczmarek and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, 2002 Springer-Verlag LNCS, Grenoble, France, April, 2002.

[Tsay2000] J. Tsay. A Code Generation Framework for Ptolemy II. *ERL Technical Memorandum UCB/ERL No. M00/25*, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000.

[Zhou2007] G. Zhou, M.-K. Leung and E. A. Lee. A Code Generation Framework for Actor-Oriented Models with Partial Evaluation. In *Proceedings of International Conference on Embedded Software and Systems 2007*, LNCS 4523, pp. 786-799, Daegu, South Korea, May 14-16, 2007, Y.-H. Lee et al. (Eds.)