Actor-oriented Metaprogramming

by

Stephen Andrew Neuendorffer

B.S. (University of Maryland, College Park) 1998
B.S. (University of Maryland, College Park) 1998
M.S. (University of California, Berkeley) 2002

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

Fall 2004

The dissertation of Stephen Andrew Neuendorffer is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2004

Actor-oriented Metaprogramming

Abstract

Actor-oriented Metaprogramming

by

Stephen Andrew Neuendorffer
Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward Lee, Chair

Robust design of concurrent systems is important in many areas of engineering, from embedded systems to scientific computing. Designing such systems using dataflow-oriented models can expose large amounts of concurrency to system implementation. Utilizing this concurrency effectively enables distributed execution and increased throughput, or reduced power usage at the same throughput. Code generation can then be used to automatically transform the design into an implementation, allowing design refactoring at the dataflow level and reduced design time over hand implementation.

This thesis focuses particularly on the benefits and disadvantages that arise when constructing models from generic, parameterized, dataflow-oriented components called *actors*. A designer can easily reuse actors in different models with different parameter values, data types, and interaction semantics. Additionally, during execution of a model actors can be reconfigured by changing their connections or assigning new parameter values. This form of reconfiguration can conveniently represent adaptive systems, systems with multiple operating modes, systems without fixed structure, and systems that control other systems. Ptolemy II is a Java-based design environment that supports the construction and execution of hierarchical, reconfigurable models using actors.

Unfortunately, allowing unconstrained reconfiguration of actors can sometimes cause problems. If a model is reconfigured, it may no longer accurately represent the system being modeled. Reconfiguration may prevent the application of static scheduling analysis to improve execution performance. In systems with data type parameters, reconfiguration may prevent static analysis of data types, eliminating an important form of error detection. In such cases, it is therefore useful to limit which parameters or structures in a model can be reconfigured, or when during execution reconfiguration can occur.

This thesis describes a reconfiguration analysis that determines when reconfiguration occurs in a hierarchical model. Given appropriate formulated constraints, the analysis can alert a designer to potential design problems. The analysis is based on a mathematical framework for approximately describing periodic points in the behavior of a model. This framework has a lattice structure that reflects the hierarchical structure of actors in a model. Because of the lattice structure of the framework, this analysis can be performed efficiently. Models of two different systems are presented where this analysis helps verify that reconfiguration does not violate the assumptions of the model.

Run-time reconfiguration of actors not only presents difficulties for a system modeler, but can also impede efficient system implementation. In order to support run-time reconfiguration of actors in Java, Ptolemy II introduces extra levels of indirection into many operations. The overhead from this indirection is incurred in all models, even if a particular model does not use reconfiguration.

In order to remove the indirection overhead, we have developed a system called Copernicus which transforms a Ptolemy II model into self-contained Java code. In performing this transformation the Java code for each actor is specialized to its usage in a particular model. As a result, indirection overhead only remains in the generated code if it is required by reconfiguration in the model. The specialization is guided by various types of static analysis, including data type analysis and analysis of reconfiguration. In certain cases, the generated code runs 100 times faster and with almost no memory allocation, compared to the same model running in a Ptolemy II simulation. For small examples, performance close to handwritten Java code has been achieved.

Professor Edward Lee
Dissertation Committee Chair

To Cynthia and our Acorn.

“I wanted a perfect ending. Now I’ve learned, the hard way, that some poems don’t rhyme, and some stories don’t have a clear beginning, middle, and end. Life is about not knowing, having to change, taking the moment and making the best of it, without knowing what’s going to happen next.” *–Gilda Radner*

Contents

List of Figures

Acknowledgments

This thesis describes a relatively small portion of the entire Ptolemy II system, which many people have put their own effort into. I have had the pleasure of working with almost everybody who has contributed to Ptolemy II. In order to avoid forgetting anyone, I will leave them largely unnamed: they know who they are. I hope I will again find a group of such motivated and interesting people.

However, there are several people who deserve special attention for their contributions. The work on Copernicus was inspired by several discussions with Jeff Tsay. Christopher Hylands Brooks provided invaluable assistance with Copernicus, including some rather nasty scripts for collecting performance data and performing treeshaking on generated code. Xiaojun Liu discovered an error in an early version of one of the proofs. Rachel Zhou was patient enough to listen to some early ideas that eventually resulted in the reconfiguration analysis. Haiyang Zheng helped push a few models through Simulink for performance comparison. Lastly, Pieter Mosterman mentioned the difficulty with naive modeling of variable capacitors.

Edward Lee has always been an open-minded and consistent supporter from the first time I walked into his office as a not-yet student. Looking back, he always seemed to know when I really needed advice and when I was better off figuring things out for myself. I also greatly appreciate the time that Kurt Keutzer, Masayoshi Tomizuka, and Tom Henzinger took from their busy schedules to sit on my qualifying exam committee and review my work. As an undergraduate, Nariman Farvardin, Dave Stewart, and the late John Gannon were valuable mentors and instrumental in encouraging me to pursue graduate work.

Lastly, my wife Cynthia has been a great source of encouragement and support. I'm sure that we will find many new adventures.

Chapter 1

Introduction

For inexperienced software engineers, writing correct programs can be a challenge. Often, simply writing syntactically valid programs that can be compiled is difficult, disregarding functional correctness, testing, or usability. However as software engineers become more experienced, addressing these issues becomes second nature and other aspects of engineering craft become more important. These engineering considerations, such as robustness, ease of maintenance, and reusability, arise in the search for better, more well-structured programs.

Although designing well-structured programs takes quite a bit of experience, recognizing such programs is relatively easy. A program that performs one task is good, but a program that performs a variety of similar tasks is generally better. A part of a program that can be extracted and reused is generally more useful than a part that cannot be extracted. A program structured according to well-described concepts or metaphors is more easily understood than one which lacks structure. A program that orthogonalizes unrelated concepts from one another is more easily modified than one with highly dependent concepts. A program with concepts that are highly localized in the program is generally less fragile than one where concepts appear in many places.

Unfortunately, although many software engineering advances help programmers to construct well-structured programs, these improvements often come with cost. For instance, common object-oriented design patterns deal well with concept localization and orthogonalization but add indirection overhead. Recursive algorithm implementations can be more concise than iterative implementations at the expense of stack usage and procedural overhead. Binary component frameworks often incur large communication overhead across component boundaries.

In contrast, highly optimized software is often highly obfuscated and more difficult to modify directly. Control structures are simplified by code duplication and explicit case expansion, requiring later changes to be made consistently in multiple places. Assumptions are made about the organization of data that must be guaranteed by code elsewhere in the program. The intent of a designer is also often obscured through the replacement of meaningful identifiers and expressions to save space.

1.1 Metaprogramming and Generative Programming

One way of approaching making a better tradeoff between organization and optimization is by *Metaprogramming* or *Generative Programming* [21], i.e., using one program (a metaprogram) to describe another. Interpreting a metaprogram, either through *execution* or through *compilation*, generates the desired program. This extra step, allows the metaprogram to be nicely organized, even when the generated program is not. The process of metaprogramming is illustrated in Figure 1.1.

Metaprogramming is often performed using pre-processor macros, allowing a programmer to automatically generate code that would be awkward or redundant to write by hand. The template mechanism in C++ is a structured, turing-complete language that is evaluated at compile time. Although awkward, the template mechanism has been used effectively to generate efficient software frameworks [21, 35]. In C++, new language features intended to directly support metaprogramming have been developed [101]. By expressing compile-time operations in a C++-like syntax and allowing richer design patterns, new language features have the potential to make metaprogramming more accessible to C++ programmers. In some cases, metaprogramming has also been found applicable to dynamic runtime code generation [19, 50].

Metaprogramming is commonly used to perform manual *program specialization* to improve execution performance. In cases where a C++ compiler might otherwise not perform certain analysis or optimizations, such as loop unrolling or propagation of constant argu-

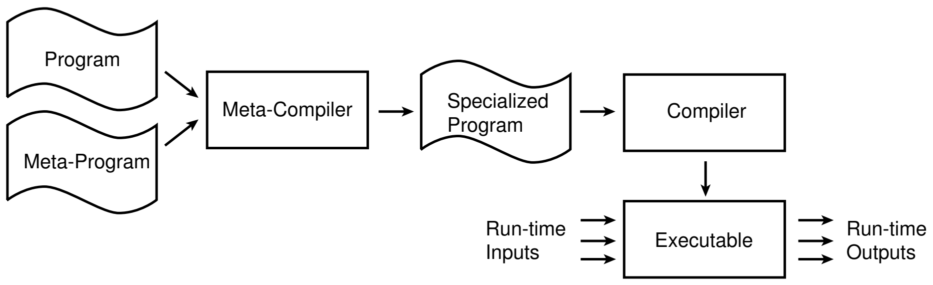


Figure 1.1: A diagram illustrating metaprogramming. The input to the meta-compiler consists of a program and a metaprogram, which are explicitly distinguished. Executing the metaprogram on the program results in a new program, which is then compiled and executed.

ments into a function body, these optimizations can be manually specified using a metaprogram. Specific optimizations that would be impractical to build into a compiler, either because they are not correct in all cases or because they are complex and rarely applicable, can be applied programmatically. Fundamentally *any* optimization that can be performed using compile-time information can be written using a metaprogram. This technique has been particularly used to build highly optimized C++ libraries for numerical computation [97].

Metaprogramming mechanisms have also been built into hardware description languages used to specify ASIC circuits or FPGA configurations. The `generate` statement in Verilog-2001 allows for automatic generation of circuit structures. The structure of these circuits must be statically elaborated early in the process of circuit synthesis. Recent work on Bluespec [40, 5, 82] is partially concerned with providing improved semantics for circuit synthesis in the presence of automatically generated circuit structures.

The above metaprogramming approaches do not tend to enforce patterns or structure in metaprograms, leaving designers responsible for building good metaprograms. *Aspect-oriented programming* provides a structured form of metaprogramming targeted at a specific architectural problem [55, 54]. In particular, aspect-oriented programming attempts to elegantly represent and encapsulate of functionality in a program which is normally distributed throughout the program. Such *cross-cutting* functionality is represented by an *aspect*. Aspects are incorporated into the primary functionality of a program by the *aspect weaver*

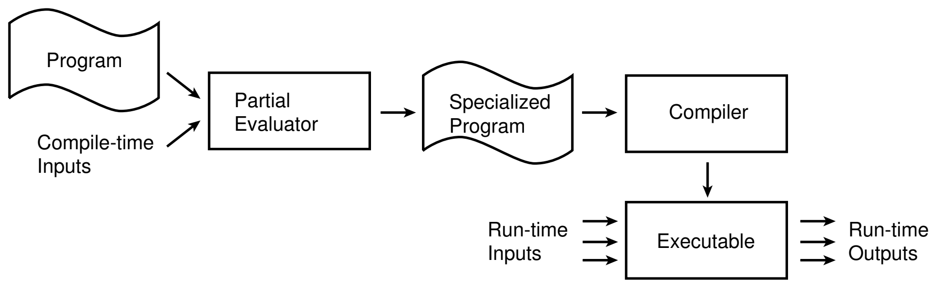


Figure 1.2: A diagram illustrating partial evaluation. The input to the partial evaluator is a program and a partial set of inputs. The resulting specialized program can be compiled and executed on the remaining inputs.

that outputs code which can be compiled normally.

Because of the separation between *design time* when compilation is performed and metaprograms are executed and *execution time* when normal system processing occurs, languages that support metaprogramming are also called *two-stage languages* or *two-level languages*. One disadvantage this separation is that a programmer must explicitly distinguish which parts of an algorithm are executed at a particular stage of execution. In other words, a program must express an algorithm differently in order to take advantage of the beneifts of metaprogramming.

One way of eliminating this separation is to focus on automatic specialization of a generic program to a particular set of inputs. This technique is called *partial evaluation* [57]. In languages with the ability to manipulate programs as well as data, such as functional languages, partial evaluation is a natural way of generating more efficient programs. The partial evaluation process is illustrated in Figure 1.2.

Recently, there has been significant interest in applying partial evaluation techniques to complete object-oriented languages [20, 89, 90, 91]. These techniques rely on sophisticated inter-procedural *binding-time analysis* to infer which functions in a program should be partially evaluated. Variables with *static binding time* can be computed by the partial evaluator and used to specialize a program. Variables with *dynamic binding time* remain in a partially evaluated program and are computed at run-time. Although effectively making use of these techniques generally requires some manual annotation of binding times in a program, partial evaluation generally requires less explicit specification by a programmer than other metaprogramming techniques.

1.2 Component Based Design

Another approach to better program organization is *component based design* [93]. Component based design focuses on component specifications with well-defined external interfaces. The external interface exposes all of the points of interaction with other components and complex behaviors can be constructed by composing components using their external interfaces. One benefit of component based design is improved *design reuse*, where components are reused from one design to the next or purchased as intellectual property from component providers [75]. Design reuse can reduce time spent in the design cycle when applied effectively [76].

One disadvantage of many component models is the overhead of run-time component interfaces. Component middleware that provides rich interactions between components, such as the Common Object Request Broker Architecture (CORBA) or the Message Passing Interface (MPI), can have significant overhead for fine-grained components. Although simpler component frameworks may have less overhead, simpler frameworks are often more constraining and difficult to leverage effectively. To make matters worse, migrating from one component framework to another, if the initial choice turns out to be unsatisfactory, often involves significant reimplementation.

Recently, compile time analysis and transformation of component frameworks in order to optimize execution has been applied to object-oriented component frameworks, particularly in embedded systems [1, 59, 77, 96, 102]. These systems use design time information to specialize a composition of components in a particular model. These systems are capable of breaking component interfaces apart in order to integrate components more efficiently and perform this composition in a safe manner, a process that can be generically called *invasive software composition* [4]. These systems preserve components as a design-time abstraction, while eliminating component interfaces at execution time.

1.3 System-level Design

Addressing the tradeoff between between organization and optimization is also important at a system-level of design. At the system level, understanding design structure becomes more critical since resource constraints and architectural tradeoffs can greatly affect the behavior of a system. Without a system-level viewpoint, these properties must be derived from low-level design, which can limit the effectiveness of system-level refactoring.

Concurrency plays an important role in system-level design. The concurrency between software running on different processors, or different threads on a single processor, is usually hidden in most programming languages. Unfortunately, current design practice tends to architect concurrent behavior through low-level mechanisms, such as monitors and critical sections. Given the flexibility and potential pitfalls in these mechanisms, a higher-level approach to managing concurrency is preferable.

Part of the system-level approach of this thesis is to consider the design of hardware, e.g., digital circuits and FPGA configurations, and software, e.g., microprocessor programs. Traditionally, hardware design has focused on concurrent, cycle-accurate, register-transfer level design, while software design has focused on sequential, behavior-accurate, function-level design. However, increasingly, this separation is fading. In hardware, there is a need to increase the speed of the design process as systems become larger. Although some design will inevitably performed at a cycle-accurate level, higher-level design techniques are still in great demand. In software, untimed and sequential programming abstractions are becoming less attractive, particularly in embedded systems which are intrinsically timed and distributed.

Recently, there has been a trend towards embedding system-level constructs into existing programming languages, as in Scenic [71] and SystemC. The resulting library is often called a *domain-specific embedded language*, or DSEL [42]. While this approach adds no fundamentally new semantics, DSELs can be significantly easier to build (since they can rather easily evolve) and for users to learn. Instead of learning new syntax and semantics, designers instead learn libraries and techniques for using these libraries to represent behavior.

One disadvantage of embedded languages is increased execution overhead, since libraries often add indirection. For domain specific languages embedded into functional languages, such as Haskell, this indirection can be removed through partial evaluation [42]. The resulting specialized program has the design time benefits of domain-specific constructs combined with execution efficiency of a low-level implementation.

1.4 Actor-oriented Metaprogramming

Actor-oriented design is an approach to system-level design using concurrent dataflow-oriented components called *actors*. Actors specify behavior abstractly without relying on low-level implementation constructs such as function calls, threads, or distributed computing infrastructure. Typically actor-oriented models are designed to reflect the static structure of a concurrent system. In this thesis, actor-oriented models are viewed as descriptions of concurrent software architectures, i.e., structured metaprograms. The behavior of the model can be *simulated* without compile-time interpretation of the metaprogram or *specialized* and executed in a more efficient manner [81]. Specialization is largely based on model analysis in the style of partial evaluation tools, rather than through explicit metaprogramming.

This thesis provides two main contributions. Firstly, it will present a formal model for analyzing reconfiguration and parameter dependencies in actor-oriented models. This model is a central part of the compile-time analysis that guides specialization of actors to particular parameter values. The model also provides a decidable algorithm for validating constraints on reconfiguration in a model which often arise from other forms of specialization, such as dataflow scheduling analysis. For this reason, analysis of reconfiguration is described as a *behavioral type theory of reconfiguration*.

The second contribution is a system for generating optimized software implementations of actor-oriented models through automatic specialization of actor oriented models. This system is constructed within Ptolemy II [44], an object-oriented software framework where components are highly generic and reusable. In Ptolemy II, generic aspects of components are implemented using run-time mechanisms, such as interfaces and indirection. Based on compile-time analysis of the model, actor-oriented components can be specialized to particular uses, reducing the need for costly run-time indirection. The resulting system can be considered in many ways: as a code generator for a concurrent programming language, as a tuned partial evaluation system, or as a component optimization system.

In contrast with other actor-oriented systems, such as Ptolemy ’Classic’ [14, 85] or commmercial tools such as Simulink from The Mathworks, the system described here allows models to be reconfigured in various ways during execution. This reconfiguration may include modification of types and the structure of the system in addition to the modification of parameter values. Furthermore, the generation of an optimized implementation incorporates code that is used for simulation as a specification of behavior. In contrast, other systems generally use separate specifications for simulation models of actors and for generating specialized code. As a result, ensuring consistency of these separate specifications becomes a significant problem. This thesis follows an approach based on systematic optimization of a single behavioral specification. This approach can also be combined with more explicit specifications of generated code for components that are not sufficiently optimized, but we anticipate that the majority of a system can be generated automatically.

Chapter 2

Actor-oriented Design

Actor-oriented modeling and design [63, 65, 67] is a methodology for system-level design that has evolved over many years of research. Carl Hewitt and others developed basic techniques for constructing systems based on *asynchronous message passing*, instead of applicative evaluation, as in the lambda calculus [38, 39]. Gul Agha developed a formal theory for describing concurrent systems that combined Hewitt's message passing with local state update [2, 3]. More recent work [29] focuses on the use of patterns of message passing between components, called *models of computation*, with interesting modeling properties. This thesis focuses particularly on *dataflow* models of computation derived from the work of Gilles Kahn [47, 48] and Jack Dennis [24].

The actor-oriented models described in this thesis differ from the actor models of Hewitt and Agha in several ways. In particular, Hewitt and Agha focus on dynamically instantiated actors and acquaintance relationships for message passing between actors. In contrast, this thesis emphasizes models with static structure and shared life cycle, while still allowing dynamic instantiation. Secondly, Hewitt and Agha view actors as a universal concept; everything in the system is an actor that responds to messages. This thesis will tend to distinguish data tokens, which encapsulate data and do not interact with one another, from actors which exchange and process data. This distinction allows the optimization of execution performance with respect to the static structure of a model without considering dynamic data.

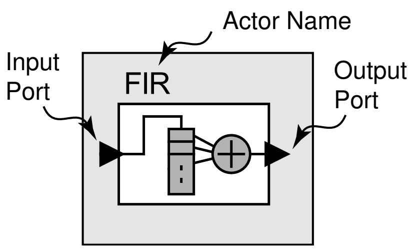


Figure 2.1: An example of the interface of a simple actor in Ptolemy II. Parameters and internal state are not shown.

2.1 Actor-oriented Models

In actor-oriented design, *actors* are the primary units of functionality. Actors have a well defined interface, which abstracts internal state and execution of an actor and restricts how an actor interacts with its environment. Externally, this interface includes *ports* that represent points of communication for an actor and *parameters* which are used to configure the behavior of an actor. Actors will be shown graphically in the Ptolemy II [44] style, as in Figure 2.1.

Actors are composed with other actors to form *composite actors* or *models*. Connections between actor ports represent communication *channels* that pass data *tokens* from one port to another. The semantics of composition, including the communication style, is determined by a *model of computation*. When necessary, the model of computation will be shown explicitly as an independent *director* object in model. Models often export an external actor interface, enabling them to be further composed with other models. A simple actor-oriented model is shown in Figure 2.2.

A central concept in actor-oriented design is that internal behavior and state of an actor are hidden behing the actor interface and not visible externally. This property of *strong encapsulation* separates the behavior of a component from the interaction of that component with other components. System architects can design at a high level of abstraction and consider the behavioral properties of different models of computation independently from the behavioral properties of components. Furthermore, different models of computation can be used at different levels of hierarchy, enabling *hierarchically heterogeneous* design [29]. By emphasizing strong encapsulation, actor-oriented design addresses the *separation*

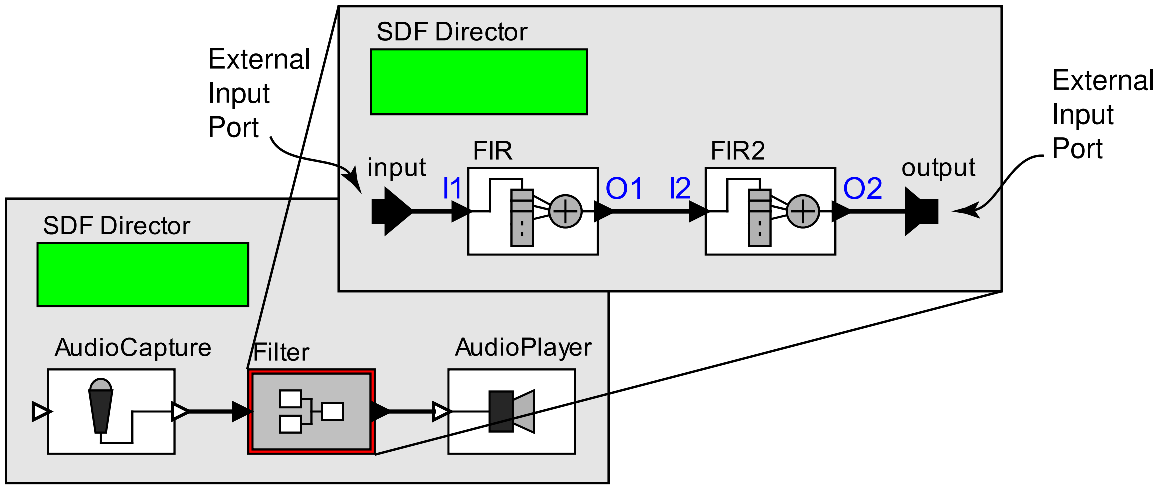


Figure 2.2: An example of a simple hierarchical synchronous dataflow model in Ptolemy II. The filter component is hierarchically decomposed into two multi-rate FIR filters with input and output token rates shown on the figure. The synchronous dataflow scheduler uses these rates to compute the number of tokens consumed and produced by the filter actor, as will be described in Section 2.6.1.

of concerns [53] between component behavior and component interaction.

In addition to supporting hierarchically heterogeneous models, strong encapsulation allows primitive or *atomic* actors to be specified in a variety of ways. For instance, actors are often specified by drawing finite-state machines where each transition corresponds to a particular sequence of operations [72]. Another technique is to use a special purpose textual language that specifies what tokens to consume and what operations to compute on that data, such as CAL [30, 100]. However, one of the most flexible ways to specify actor behavior is to embed the specification within a traditional programming language, such as Java or C, and use special purpose programming interfaces for specifying ports and sending and receiving data. This technique has been widely used in actor-oriented systems [15, 31, 80] since it allows for existing code to be integrated into an actor-oriented design tool and for programmers to quickly start using actor-oriented methodologies.

2.2 Hierarchical Semantics

Previous work has focused on giving a formal framework for describing the behavior of actors and models as transition systems [39, 45, 73]. Transitions represent both internal computation and external interaction, such as the production and consumption of data from the ports of an actor or coordination with other actors. The behavior of an actor is primarily determined by an *actor specification*. The execution of this specification in the context of other actors and data from the environment results in an actor's observed behavior.

In this framework, a model of computation determines a style of composition of actor specifications. This style includes the behavior of communication channels between actors, and any additional control logic for ordering the transitions of actor specifications. As with actors, communication and control logic can also be represented as transition systems. Composing the transition systems for individual actors in a model with the transition systems representing a model of computation results in a transition system for the entire model.

In order to be composed in this framework, actors and models are required to have *precise reactions* [73]. Each actor must consist of a totally ordered sequence of *actor firings* or *iterations*, which are similar to concept of an *activity* developed by Hewitt and Baker [39]. During the firing of an actor, it may send and receive data from communication channels and perform computation. Between firings, an actor is *quiescent* and cannot communicate or perform computation. Additionally, compositions are required to be *hierarchically reactive*, where each actor firing must be encompassed by a single firing of the actor's container. Equivalently, when a composite actor is quiescent, all actors deeply contained by the composite actor are also quiescent. A figure illustrating the quiescent points of a model is shown in Figure 2.3.

Because the internal state of each actor is hidden from other actors in a model, there are few intrinsic constraints on the firings of different actors in a model. The lack of execution constraints implies that actors in a model are *fundamentally concurrent*. In order to obtain efficient execution in single processor environments, additional execution constraints are often added to a model to enforce sequential execution of actor firings. In the absence of additional constraints, the firings of two distinct actors are allowed to occur completely

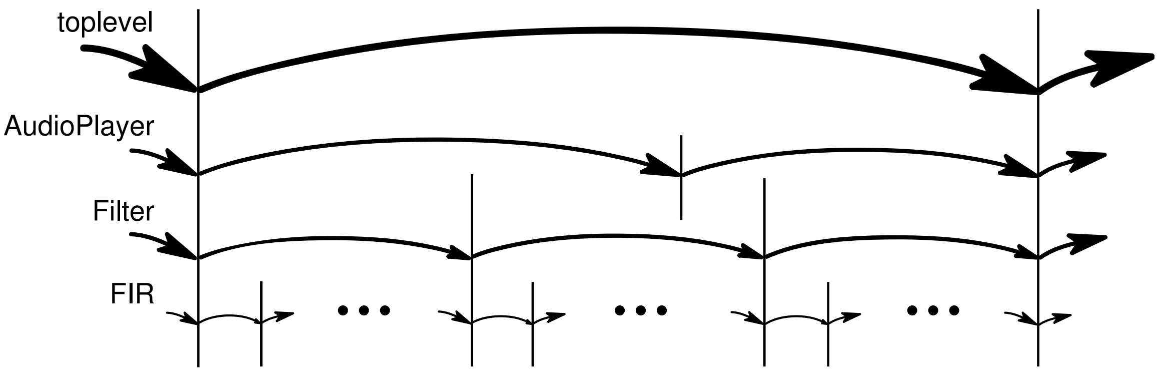


Figure 2.3: A graphical representation of the quiescent points in one execution of the model in Figure 2.2. The model is one where actor toplevel contains actors AudioPlayer and Filter, and actor Filter in turn contains actor FIR. Quiescent points are shown as vertical lines and actor firings are shown as arrows. A quiescent point is a quiescent point of an actor if a firing arrow of the actor starts or end at the quiescent point. The direction of arrows represents the partial ordering of quiescent points.

independently.

Unfortunately, the fundamentally concurrent nature of actor-oriented models can result in rather complex transition systems for actor compositions. In contrast, the structure of quiescent points in a model is somewhat simpler. Quiescent points abstract the concurrency and sequentiality between actors in a model and represent only the hierarchical relationships between actors. This simpler framework for formal analysis can be useful when analyzing hierarchical properties of models, as will be seen in Chapter 4.

2.3 Parameterization and Reconfiguration

The communication interface consisting of an actor's ports also allows actors to be developed independently and provided as reusable library elements. Actor parameters increase the reusability of such library elements, by allowing the same actor specification to be reused with different parameter values. For instance, an actor representing a finite-impulse response (FIR) filter might have a parameter that determines the filter taps. The same actor might also provide multi-rate capabilities for efficient upsampling and downsampling, with corresponding parameters to determine the number of tokens produced and consumed during each execution of the filter. At design time, parameters help keep the size of actor libraries manageable and allow models to be quickly modified or tuned for performance. At run time, actor parameters allow for dynamic reconfiguration of actors (and models) while a model is executing.

There are many applications that can make use of dynamically reconfigured models. For instance, a communication system with adaptive echo cancellation can be modeled by reconfiguration of a parameterized filter. Parameter reconfiguration enables a single component to be used either as a fixed filter or an adaptive filter, obviating the need for two separate components. At a coarser level of granularity, systems with multiple operating modes are common. For instance, a communication system might operate in either a training mode or a communication mode. In the training mode, the system communicates a predetermined bit sequence and estimates the characteristics of the channel. These characteristics are used in the communication mode to improve the bit-error performance of the modem. Transitions between training mode and communication mode can be modeled as system reconfiguration.

In actor-oriented models, parameters are usually used to represent configuration values specified by a designer. Configuration parameters are usually used in a *top-down* fashion, where parameter values of contained actors are dependent on parameter values higher in a model. However, parameters in actor-oriented models can also be used to represent synthesized properties of actors, such as data types, execution schedules, and token rates. The dependencies between such parameters are generally *bottom-up*, with parameters of the toplevel depending on parameters of contained actors. In this thesis we will leverage parameters as a uniform representation for properties in a model that can be affected by reconfiguration.

We distinguish two forms of reconfiguration: parameter reconfiguration and structural reconfiguration. Parameter reconfiguration changes the value of actor parameters, while structural reconfiguration may add or remove actors and modify the connections between ports. In actor-oriented models, both structural and parameter reconfiguration are allowed only at quiescent points in execution. The hierarchical structure of quiescent points can be used to analyze parameter reconfiguration for statically structured models, as described in Chapter 4.

2.4 Dataflow Models of Computation

In actor-oriented models, many models of computation are possible [63, 66]. By using different models of computation, actor-oriented modeling principles can be adapted to a variety of problem domains, such as physical systems, communication networks, and embedded systems. For the purposes of this thesis, we focus on dataflow models of computation applied to computing systems.

Dataflow models of computation [24, 47, 69] have been used to represent a wide variety of computing systems, such as signal processing algorithms [85], distributed computing workflows [74, 99], and embedded processing architectures [49, 92]. In a dataflow model, message communication between actors is performed through queues of data. These message queues desynchronize the communication between actors, allowing the sending actor to continue concurrently without waiting for the message to be received. At the same time, message queues ensure that messages are received in order of transmission with no message loss.

Dataflow models of computation are appealing since they often closely match a designer's conceptualization of a system as a block diagram. Additionally, dataflow models of computation offer opportunities for efficient concurrent and sequential implementation. Since actors only communicate through ports and do not share state, system parallelism is exposed in the model and concurrent execution is possible. However, parallel implementation is not required and static scheduling analysis can generate efficient sequential implementation.

2.5 Dataflow Execution

One advantage of dataflow modeling is a wide variety of techniques for operationally executing a model. One of the most basic execution techniques, known as dynamically scheduled dataflow (DDF), requires actors to declare their *token rates* for each input and

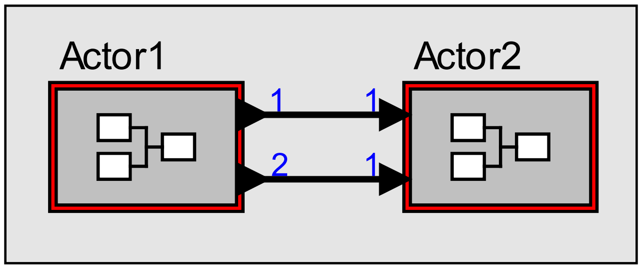


Figure 2.4: A dataflow model that cannot be executed in bounded memory.

output port. The token rate of a port determines the number of tokens that port will produce or consume during the next firing of the actor. Based on this information, a centralized scheduler decides on the next actor (or actors) to fire. After firing those actors, the scheduler inspects the token rates which may have changed, and selects another set of actors to execute.

A closely related technique for executing dataflow models, the process network (PN) model of computation [48], does not rely on a centralized dataflow scheduler or declarations of token rates. Instead, each actor in a process network is associated with a independent sequential thread of control, roughly corresponding to an operating system process. The actor's thread of control processes data from input ports as it becomes available using a *blocking read*. If no input is present, an actor must block until the data is available and there can be no way for an actor to query for the presence of data. In a process network, blocking reads ensure that the output of a dataflow model is correct, regardless of when processes are actually executed. As a result, operating system scheduling techniques that are not aware of token rates can be applied in order to execute more than one actor on a single processor.

In general, dataflow models of computation are *Turing complete* and it is undecidable whether a particular dataflow model can be executed in bounded memory. Both DDF and PN are capable of executing arbitrary dataflow models, even those that require an unbounded amount of memory, such as the model in Figure 2.4. However, they can be implemented robustly so that every model which can execute in bounded memory will actually be executed in bounded memory, using Park's algorithm [69, 83, 7].

2.6 Static Dataflow Scheduling

Although dataflow models can require an unbounded amount of buffer memory in order to execute, many interesting computations can be executed in bounded memory. Although showing this for an arbitrary model is undecidable, model analysis techniques have been developed that allow bounded memory usage to be guaranteed for restricted forms of dataflow models. These techniques determine if a given dataflow model has a *minimal complete cycle* [16], which is a sequence of actor firings that returns the dataflow model to the same original state without deadlocking. If complete cycles do exist, then dataflow scheduling techniques are usually capable of producing a finite length *execution schedule*, which results in complete cycles when executed.

Unfortunately, even if execution schedules can be found, they may not have desirable properties for system design. In particular, a schedule may use an unbounded amount of buffer memory before returning to the original state. It is also possible for a schedule to use a bounded amount of buffer memory, but require an unbounded number of actor firings. This section will summarize these dataflow scheduling techniques, and summarize the conditions under which execution schedules can be found which are *finite length*, *bounded memory*, and can be executed in *bounded time*.

2.6.1 Synchronous Dataflow

In a synchronous dataflow (SDF) model [11, 64], the token rates of actors are assumed to not change during execution. Given the token rates, an execution schedule consisting of a finite sequence of actor firings can always be found, if a minimal complete cycle exists. The resulting scheduling is guaranteed to use bounded buffer memory and to produce bounded executions. Since buffer memory usage is bounded, the synchronous dataflow model of computation is not Turing complete, unless actors can use unbounded memory internally.

SDF analysis occurs in two steps. The first step involves solving a set of *balance equations* determined by the token rates and structure of the model. The balance equations require that the number of tokens produced on a channel match the number of tokens consumed. Generally, these equations will have a set of linearly dependent solutions that determine the number of times each actor must fire in a complete cycle. Selecting the smallest positive solution gives the shortest schedule, although any positive solution can be used. If such a solution exists, an execution schedule is possible. However, if no non-zero solution to the balance equations exists, then the model is *inconsistent* and cannot be executed forever in bounded memory.

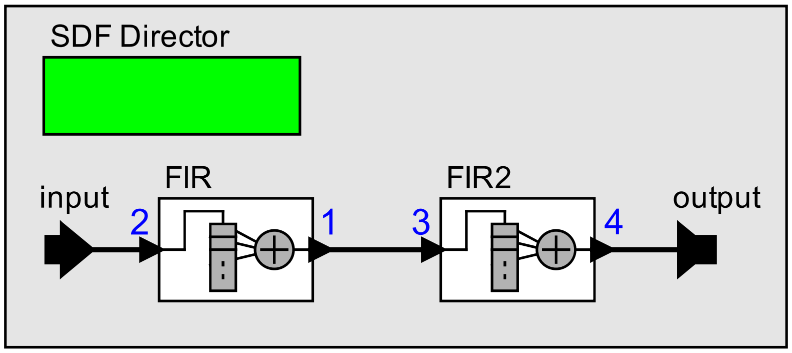
The second scheduling step simulates the execution of the model, allowing each actor to fire a maximum number of times determined by the balance equations. If the simulated execution does not deadlock, then the resulting sequence of actor firings is an execution schedule. If an execution schedule is found, then it is guaranteed to execute in bounded memory and with a bounded number of actor firings. A scheduling example is shown in Figure 2.5.

2.6.2 Parameterized Synchronous Dataflow

Parameterized synchronous dataflow (PSDF) [8, 9] is another technique for analyzing dataflow models that allows more general models than SDF. The key difference is that parameterized synchronous dataflow allows token rates to change during execution. However, rate changes are only allowed between executions of a parameterized schedule. Such a model is called *locally synchronous* [10]. For locally synchronous models with bounded token rates, PSDF schedules are guaranteed to use bounded memory and have bounded executions.

The procedure for parameterized synchronous dataflow scheduling is very similar to synchronous dataflow scheduling, except that token rates are considered to be variables, instead of constants. The balance equations are solved symbolically, and a quasi-static schedule is generated that is a function of the token rates determined at run-time. Although the execution of this schedule depends on token rates that might change at run-time, the schedule is statically determined and can be compiled into efficient executable software.

In Figure 2.6, a solution to the balance equations exists for any token rates. Such a model is called *strongly consistent* [62]. Alternatively, it is possible for some models that an integer solution to the balance equations only exists for certain token rates, in which case the model is only *weakly consistent*. As with SDF, is it also possible for no non-trivial solution to exist, in which case the model is *inconsistent* and no execution schedule can be



(a) Example Model

$$\begin{aligned}
\text{input.tokenConsumptionRate} &= 2 * \text{FIR.firingCount} \\
1 * \text{FIR.firingCount} &= 3 * \text{FIR2.firingCount} \\
4 * \text{FIR2.firingCount} &= \text{output.tokenProductionRate}
\end{aligned}$$

(b) Balance Equations

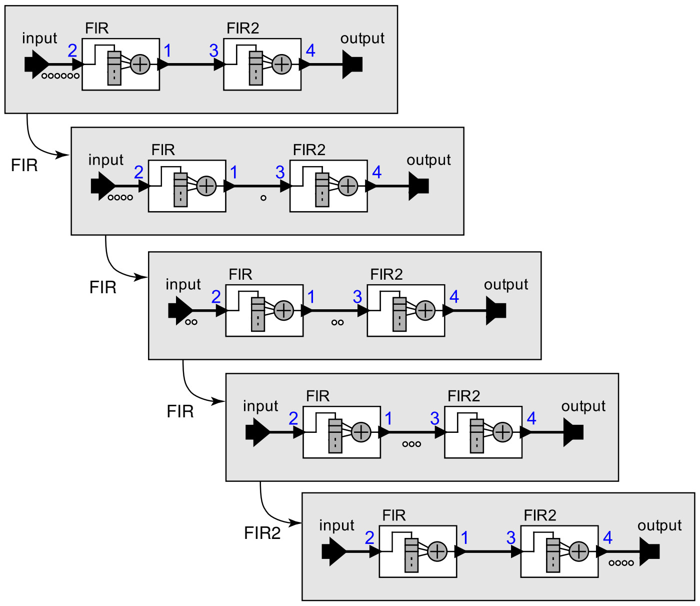
$$\begin{aligned}
\text{FIR.firingCount} &= 3 \\
\text{FIR2.firingCount} &= 1 \\
\text{input.tokenConsumptionRate} &= 6 \\
\text{output.tokenProductionRate} &= 4
\end{aligned}$$

(c) Balance Equation Solution

Figure 2.5: An example of synchronous dataflow scheduling. The balance equations for the model in (a) are shown in (b) and the least positive solution is shown in (c). Note that the rates of external ports are inferred from the solution to the balance equations.

found.

For complex parameter constraints determined by arithmetic expressions, it is generally undecidable whether a model is inconsistent, strongly consistent, or weakly consistent. Since dataflow scheduling analysis is used to assert safety properties, such as bounded memory usage, it is preferable to disallow scheduling for models which cannot be shown to be strongly consistent. However, by adding the appropriate assumptions to a model, it is often possible to manipulate a weakly consistent model into a strongly consistent one. This addition may be necessary because a parameter value is known to a designer to never



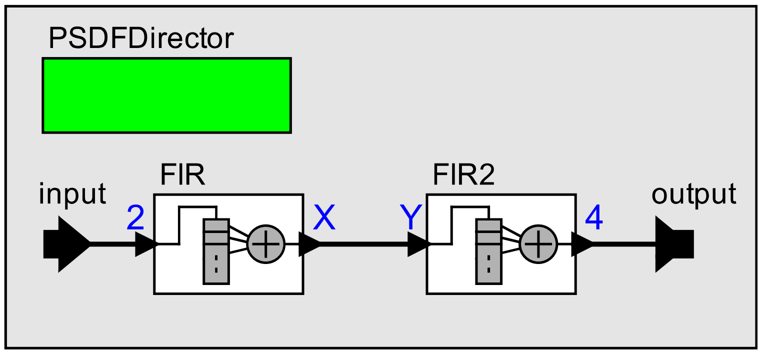
(d) Execution Simulation

Figure 2.5: Execution of the model in (a), given the required six input tokens.

change, is known to take on only constrained values, or is known to be related to other parameter values. The extra assumptions may be either checked at run-time, or become additional proof obligations in a formal analysis framework.

In PSDF models, the memory usage of a schedule generally depends on parameter values specified at run-time. Larger parameter values often result in larger memory usage (as in Figure 2.6). Hence bounded memory usage is only generally guaranteed given either constraints on parameter values or explicit bounds on buffer sizes from which constraints on parameter values can be inferred. These parameter value constraints are also treated as extra assumptions, which must either be checked at run-time or proved satisfied outside of PSDF scheduling. PSDF models are also guaranteed to have a complete cycle that execute in bounded time as a function of parameter values. If the parameter values are themselves bounded, then the resulting schedule will execute in bounded time. Based on these results, locally synchronous and strongly consistent dataflow models with bounded parameter values cannot express arbitrary computations and are fundamentally decidable.

Existing scheduling techniques based on parameterized acyclic pairwise grouping of



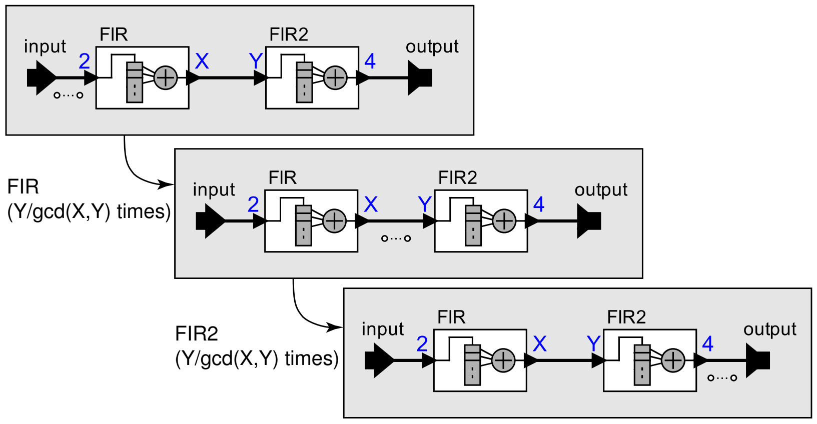
(a)

$$\begin{aligned}
\text{input.tokenConsumptionRate} &= 2 * \text{FIR.firingCount} \\
X * \text{FIR.firingCount} &= Y * \text{FIR2.firingCount} \\
4 * \text{FIR2.firingCount} &= \text{output.tokenProductionRate}
\end{aligned}$$

(b) Balance Equations

$$\begin{aligned}
\text{FIR.firingCount} &= Y/gcd(X,Y) \\
\text{FIR2.firingCount} &= X/gcd(X,Y) \\
\text{input.tokenConsumptionRate} &= 2 * Y/gcd(X,Y) \\
\text{output.tokenProductionRate} &= 4 * X/gcd(X,Y)
\end{aligned}$$

(c) Balance Equation Solution



(d) Execution Simulation

Figure 2.6: An example of parameterized synchronous dataflow scheduling.

adjacent nodes (P-APGAN) [8] generate finite schedules only for models without feedback, or models where feedback does not affect scheduling. For models with tight feedback, good static scheduling techniques have yet to be developed, although bounded run-time scheduling is still possible. It is likely that improved clustering techniques, such as those applicable to Boolean-controlled dataflow are capable of generating schedules for arbitrary PSDF models.

2.6.3 Boolean- and Integer-controlled Dataflow

Boolean-controlled dataflow (BDF) [16] and integer-controlled dataflow (IDF) [17] are closely related scheduling techniques. The primary difference between them and PSDF is that BDF and IDF lift the restriction on local synchrony and allow token rates to change every time an actor fires. This seemingly simple change makes scheduling significantly more complex, and introduces the possibility that execution schedules may not terminate. For general BDF and IDF models, determining if a model can be executed in bounded memory is undecidable, although heuristic techniques are able to find execution schedules for commonly used structures.

BDF and IDF models are usually constructed using flow control actors, shown in Figure 2.7. These actors change their behavior according to the current value of the control signal. Each token received from a control input reconfigures the actor, determining the routing of the next data token. These actors are not locally synchronous and cannot be used directly in PSDF models. In terms of the FIR filter example given previously, IDF allows the filter to have input ports which determine decimation and interpolation factors.

Switch and Select actors are usually used in well-behaved patterns or schema, such as conditionals and loops. Using these patterns, finite execution schedules can be found and complete cycles are guaranteed to be bounded [33]. However, in general, strongly consistent models are not guaranteed to execute in bounded memory, or in finite time [16]. For example, Figure 2.8 is a model that might require unbounded memory and unbounded time to execute. It is also possible for a model for model to execute in unbounded time, but only use bounded memory, as shown in Figure 2.9. In both of these models the ControlSource actor can produce an arbitrary sequence of control values, but the minimal complete cycle

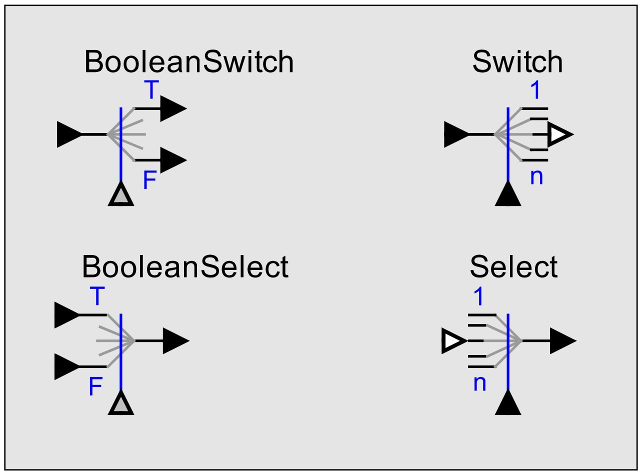


Figure 2.7: Control flow actors in Ptolemy II. BooleanSwitch and BooleanSelect take a boolean control input that determines which output consumes or produces the next token and no tokens are produced or consumed from the remaining port. Switch and Select take an integer control input that determines which channel of the multiport produces or consumes the next token and no tokens are produced or consumed from other channels. Note that the names here used in Ptolemy II differ slightly from the literature [16, 24], where actors with boolean control input are called Switch and Select and actors with integer control inputs are called Case and EndCase.

occurs only after the ControlSource actor produces two `TRUE` tokens.

Note that the scheduling difficulties of BDF and IDF models only arise in models that are not locally synchronous. If the control flow actors always receive the same value from their control ports and this fact is available to a scheduler, then SDF scheduling can be applied. If the value received from their control port doesn't change until the end of the minimal complete cycle, then PSDF scheduling can be applied. For this reason, it is useful to consider the the control ports of control-flow actors to be *reconfiguration ports*, as described in Section 3.1.

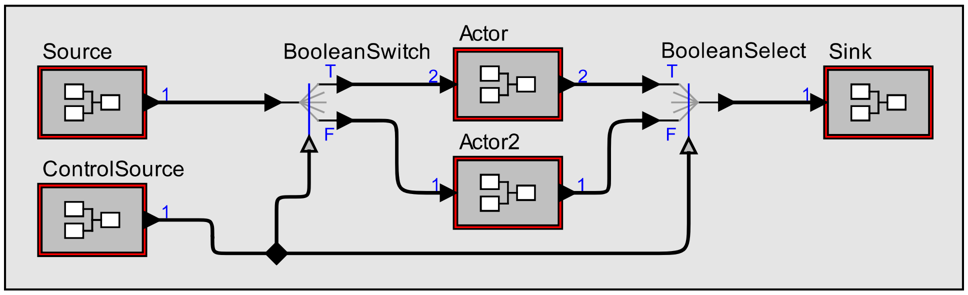


Figure 2.8: A dataflow model where a minimal complete cycle might require unbounded memory and unbounded time. In an execution where ControlSource produces a single TRUE token followed by a sequence of FALSE tokens, the output of Actor2 will accumulate at the input to the BooleanSelect actor.

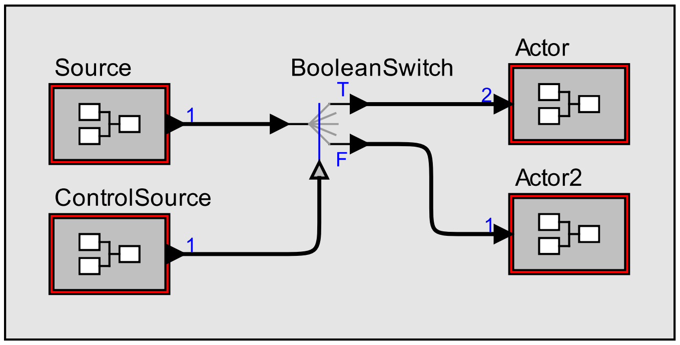


Figure 2.9: A dataflow model where a minimal complete cycle might require unbounded time, but is guaranteed to require only bounded memory.

2.6.4 Hierarchical Dataflow Scheduling

In a hierarchical actor-oriented framework, it is natural to use different scheduling techniques at different layers of hierarchy. While dataflow scheduling techniques are not typically presented in a hierarchical framework, it is generally straightforward to extend them to operate hierarchically. One possibility is to recursively schedule a hierarchical model beginning with scheduling of models deep in the hierarchy. Scheduling each model determines the token rates of external ports, which can then be used to schedule higher-level models. This procedure maximizes use of the rate information of primitive actors without additional annotation.

Declaration of token rates does not occur solely at compile-time, but may also occur at run-time when a model is reconfigured. As a result, there are dependencies between token rates that behave in exactly the same way as dependencies between configuration parameters. As mentioned previously, this fact will be leveraged by representing token rates as *rate parameters*, despite the fact that rate parameters are not usually configured directly by a designer. As a result, token rate changes can be considered in the same way as any other parameter change.

Chapter 3

Reconfiguration of Actor-oriented Models

Although actor-oriented design encourages the use of parameters and reconfiguration of a model, reconfiguration can also cause difficulties. In some cases, the implementation of an actor assumes that certain parameters don't change during execution of the actor. In other cases, reconfiguration can modify parameters that are not meaningful to modify during execution of a model, such as parameters specifying physical parameters. Reconfiguration can also indirectly affect properties of an actor or model used for static analysis, such as dataflow scheduling or type checking.

3.1 Hierarchical Parameter Reconfiguration

Many different modeling syntaxes have been used to represent reconfiguration in dataflow systems. This section presents a brief summary of the mechanisms which have been implemented in Ptolemy II.

3.1.1 Modal Models

One syntax for specifying reconfiguration is based on an extended version of a finite state machine, called a *modal model*. Each state of the finite state machine contains a dataflow model that is *active* in that particular state. The active dataflow model is alternatively called a *refinement* of the state. Essentially, the active dataflow model replaces the finite state machine until the state machine makes a state transition. Additionally, finite state machine transitions are capable of reconfiguring parameters of the target state's refinement. This syntax is similar to the *-charts model [34, 61], the FunState model [92], and the Stream-Based Functions model [56].

During each firing of a modal model, the dataflow model associated with the active state is fired once and it communicates directly with the external ports of the modal model. After the active dataflow model is fired, the guard of each transition originating in the active state is evaluated. If exactly one guard is satisfied, then that transition is taken and the destination state of the transition will be active in the next firing. If no guard is satisfied, then the active state will remain active in the next firing. If multiple guards are satisfied, then either the model is considered incorrect or one of the transitions can be chosen non-deterministically. If a transition is taken then the action of the transition is performed, possibly resulting in reconfiguration of a model parameter at the quiescent point after the firing. An example model is shown in Figure 3.1 and a plot from running the model in Figure 3.2.

Another example of a modal model is shown in Figure 3.3. This modal model has multiple states, and a different refinement model is active in each state. Each refinement is a statically scheduled synchronous dataflow model, but the external rates of the refinement models are different. When the modal model changes state, the rate parameters of the modal model are reconfigured to reflect the rate parameters of the new refinement.

Modal models (and finite state machines in general) are practically limited by the number of states that a designer can specify explicitly. It is common to use various forms of extended state machine formalisms to reduce the states that must be specified explicitly. For instance, the state machine may include state variables which may be modified in state transitions and used to govern transition guards. Another possibility is to use hierarchical state machines and parallel state machine composition, as in Statecharts [36, 37].

3.1.2 Reconfiguration Ports

The second syntax ties reconfiguration to dataflow in a model. Reconfiguration in this model is represented by *reconfiguration ports*, a special form of dataflow input port. An

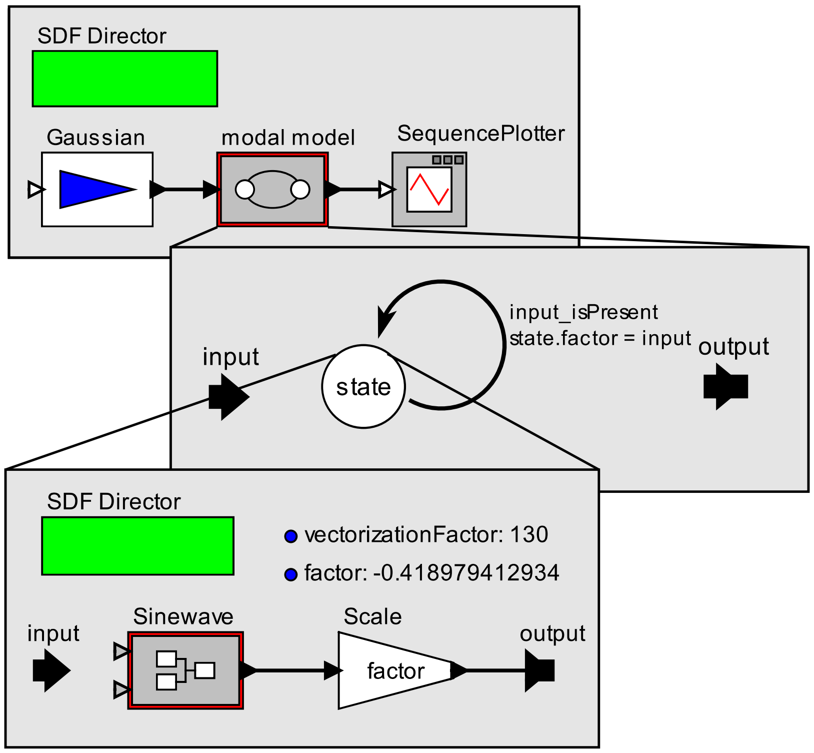


Figure 3.1: A graphical representation of a simple modal model in Ptolemy II showing three levels of hierarchy. In this model, the refinement for the current state is executed first producing a block of output tokens. After producing output tokens, the modal model transition is taken since the guard expression always evaluates to TRUE, resulting in reconfiguration of the contained model for the next block. In this model, reconfiguration results in sinusoidal segments with different amplitudes. The parameters of the interior dataflow model ensure that 130 samples of the sinusoid are generated in each block.

example of this syntax is shown in Figure 3.4. Each reconfiguration port is bound to a parameter of its actor and tokens received through the port reconfigure the parameter. More specifically, a firing of an actor with reconfiguration ports is composed of two distinct sub-firings separated by an internal quiescent state. During the first sub-firing the actor consumes a single input token only from reconfiguration ports. The input tokens determine the reconfiguration applied during the internal quiescent state. During the second sub-firing in-

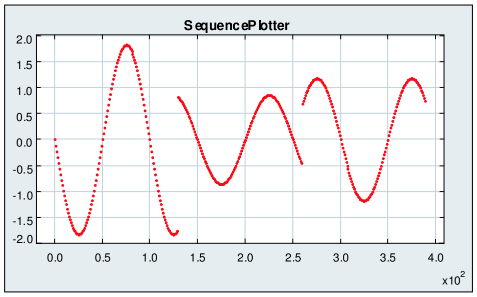


Figure 3.2: A plot from running the model shown in Figure 3.1. Note the obvious jumps in the generated signal, corresponding to mode switches in the model.

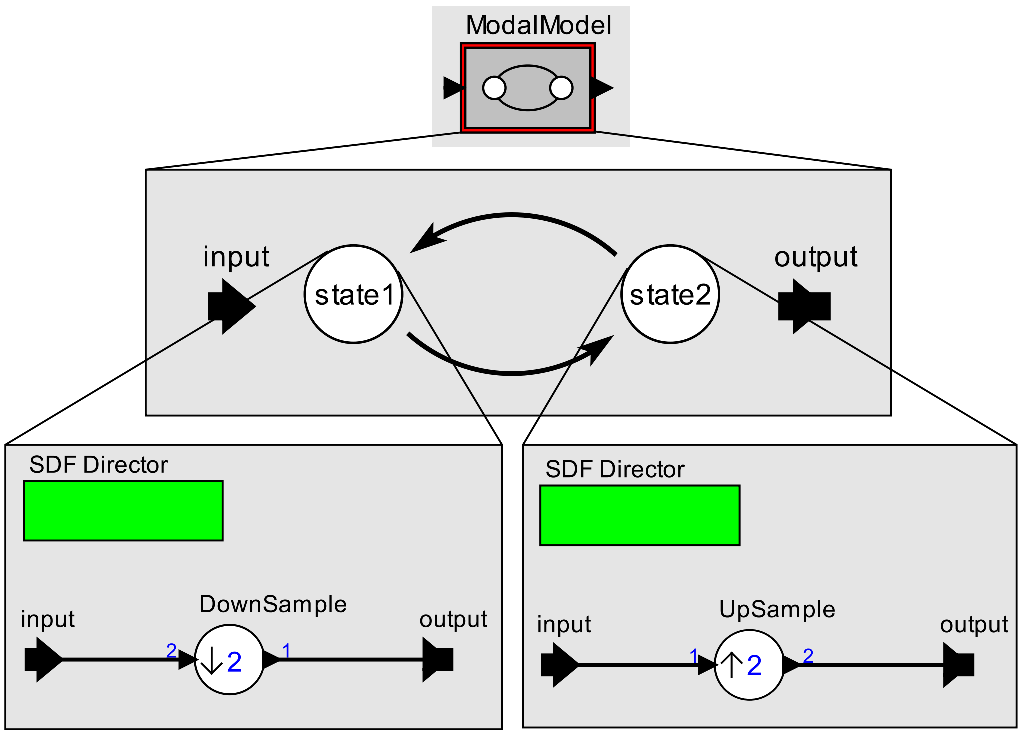


Figure 3.3: A modal model example with multiple states. In each state, the modal model reconfigures its rate parameters to reflect the rate parameters of the current refinement.

put tokens are consumed from normal dataflow input ports, computation is performed, and any outputs are produced. For a composite actor, contained actors are not fired during the first sub-firing and the associated dataflow model is executed only during the second sub-

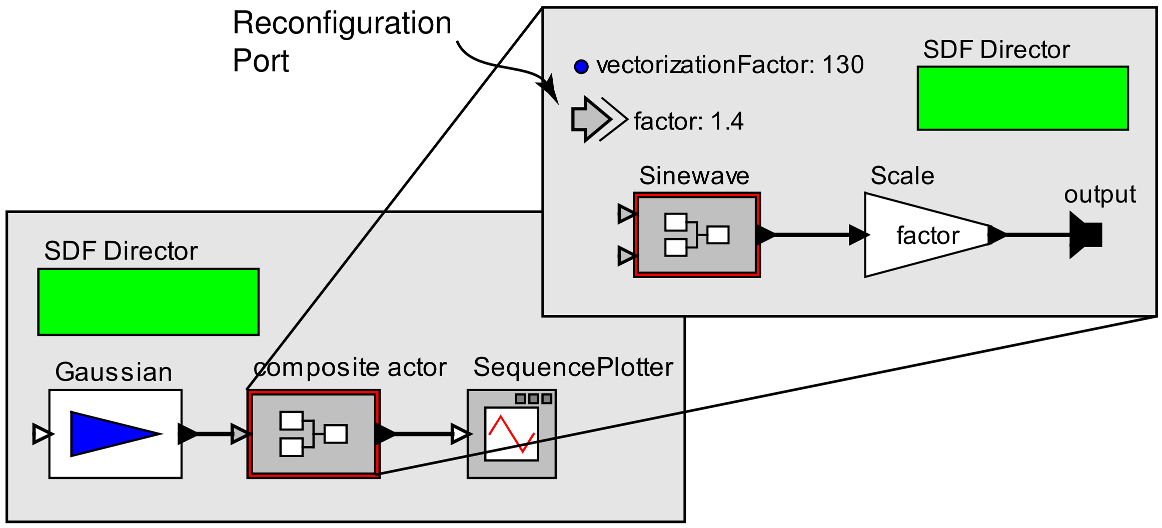


Figure 3.4: A graphical representation of a simple model with a reconfiguration port in Ptolemy II. In this model, the reconfiguration port is shaded gray instead of black and reconfigures the parameter named "factor" directly to its right. This model behaves essentially identically to the one in Figure 3.1, except that the reconfiguration occurs prior to each block of samples being produced rather than after.

firing. Reconfiguration ports exist in many dynamically-scheduled dataflow environments, such as AVS/Express (Advanced Visual Systems, Inc.).

3.1.3 Reconfiguration Actors

A third syntax represents reconfiguration using a special actor, the setVariable actor. This actor has a single input port and is associated with a parameter of the containing model. The actor consumes a single token during each firing and reconfigures the associated parameter during the quiescent point after the firing. Although the setVariable actor might appear similar to a reconfiguration port, it allows for a parameter to be more frequently reconfigured, since the setVariable actor might fire more than once in the execution schedule of its contained model.

As with reconfiguration ports, the setVariable actor can be used to implement models that are not locally synchronous. Furthermore, the setVariable actor can be used to implement models which are not deterministic. For instance, the setVariable actor can be used similarly to the graph variables in the Process Graph Method (PGM) [51]. In such models the behavior of a model is dependent on the order in which actor firings are scheduled. Although non-deterministic models are sometimes useful, they can be more difficult to design and test robustly.

3.2 Delayed Reconfiguration

One way of ensuring local synchrony in the presence of reconfiguration is to delay reconfiguration until the next quiescent point of the toplevel model. Essentially, this results in treating the above syntaxes as *requests* for reconfiguration, rather than as immediate operations. Delayed reconfiguration ensures that all parameters are constant over firings the toplevel model, and hence over the firings of any actor. However, requirements that parameters are constant must still be checked.

Delayed reconfiguration can be reasonably used in conjunction with any of the above syntaxes. Heterochronous dataflow [34] combines delayed reconfiguration with modal models and synchronous dataflow scheduling. Delayed reconfiguration combined with the setVariable actor gives a more useful mechanism for reconfiguring parameters that affect token rates since, in the absence of other reconfiguration, the model will be locally synchronous. In order to encourage the construction of safe and deterministic models, the default operating mode of the setVariable actor in Ptolemy II actually performs delayed reconfiguration.

The key disadvantage of delayed reconfiguration is that it results in models that are not *compositional*. The behavior of an actor constructed using delayed reconfiguration depends on the model in which it is placed. This situation is shown in Figure 3.5, where the setVariable actor performs delayed reconfiguration. The displayed sequence of tokens changes depending on the rate of the DownSample actor. As a result, delayed reconfiguration can be difficult to apply in reusable actor specifications.

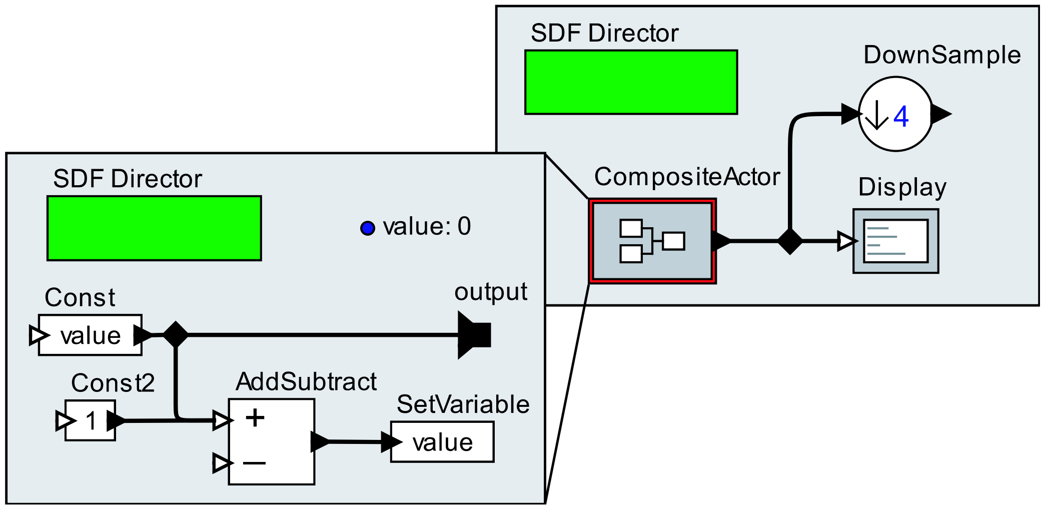


Figure 3.5: An example of delayed reconfiguration illustrated with the setVariable actor. The model displays a sequence of natural numbers. Each number is displayed a number of times given by the rate of the DownSample actor.

3.3 Efficient Parameter Evaluation

In models without reconfiguration, the computational complexity of determining parameter values is usually unimportant. Even given complex specifications in terms of other parameters, parameter values can generally be determined at design time and this evaluation does not affect run-time performance. In the presence of reconfiguration, however, evaluation of parameter values does incur run-time overhead. Minimizing this overhead is important for efficient system execution.

One general model for representing interdependent parameters is an *attribute grammar* [25, 58], commonly used to model language compilers operating on Abstract Syntax Trees (ASTs). In an attribute grammar, each terminal and non-terminal node in a syntax tree is labeled with *attributes*, according to rules given in the attribute grammar. Each attribute has an associated value, which may depend on the value of other attributes. The dependence on other attributes is determined by the *constraint function* that defines the attribute. An attribute grammar is *evaluated* by repeatedly selecting attributes which have not been assigned a value and evaluating their constraint function.

In a general attribute grammar, several modifications can affect the value of attributes. Attributes may be added or removed, dependencies between attributes may be added or removed, and the value of an attribute with no dependencies can be modified. In the presence of such modifications, incremental evaluation of attributes is desirable, since modifications often affect only a small number of attributes. A standard algorithm to evaluate attribute grammars [43] has been developed which guarantees that attribute values are evaluated only when necessary. In this algorithm, attributes are associated with an additional flag that keeps track of whether the attribute value is *valid*. When modification to an attribute grammar occurs, the flag is set for any attribute whose value may have been affected. Attribute values are recomputed in a demand-driven manner when an attribute value is required and the current value is not valid.

The common usage of parameters in actor-oriented models is somewhat simpler than arbitrary attribute grammars. While attribute grammars are normally considered for context-free languages with possibly unbounded syntax trees, hierarchical actor-oriented models are typically bounded at design time. Furthermore, the structure of actor-oriented models and dependencies between parameters are often fixed at design time. Complex computations are represented by actor interactions and parameter reconfiguration, rather than through structural reconfiguration.

3.4 Assumptions about Reconfiguration

A crucial part of the design process is to ensure that assumptions about the use of reconfiguration are met, implying that reconfiguration is used safely. Unfortunately, assumptions about reconfiguration are difficult for a designer to consider, since the effects of reconfiguration often cross levels of hierarchy in a model. Making the problem even worse, assumptions about reconfiguration are often left implicit in a model, making it difficult to check these assumptions through inspection. This section describes some examples of reconfiguration assumptions and the kinds of conflicts that can arise.

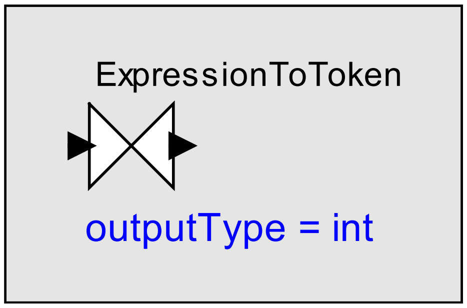


Figure 3.6: The ExpressionToToken actor, showing the parameter that determines the type of the output.

3.4.1 Reconfiguration and Type Checking

One powerful mechanism for building generic components involves the use of *type parameters*. Type parameters can be interacted with just like other parameters, but their value is used to perform type inference and static checking. Type correctness of the model cannot be guaranteed if type parameters are reconfigured during execution.

As an example, consider the ExpressionToToken actor, shown in Figure 3.6. This actor consumes a string, parses it as a parameter expression, and outputs the resulting value. For an arbitrary string, this value may be of any type. In practice, however, the type produced is known to a designer, but is not visible to the type checking mechanism. A type parameter is used to declare the type of this output.

3.4.2 Reconfiguration and Structural Parameters

In many cases is it useful to build parameterized structures in actor-oriented models. Such programmatically generated structures are called *higher-order components* to emphasize their similarity to higher-order functions in functional languages [86]. A parameter which is used to determine the structure of a higher-order component is a *structural parameter*.

The MultiInstanceComposite actor in Ptolemy II is one example of a simple higher-order component. Just before a model is executed, this actor replicates itself a number of times determined by a structural parameter. This actor is often used in situations where a model contains repetitive structures that are awkward to build by hand, or when the number of repetitions is specified by a parameter.

Although not required from a purely behavioral perspective, implementations can often benefit if structural parameters are not reconfigured. The structural parameters can be evaluated at design time to determine an equivalent actor-oriented model. The equivalent model can then be statically analyzed and optimized as if constructed manually. Such optimizations are particularly important in the hardware design community, as when synthesizing FPGAs from dataflow models [13, 28].

3.4.3 Reconfiguration and Model Correctness

In some cases, parameter reconfiguration violates the assumptions made in constructing a model. Some models are no longer accurate if certain parameters are reconfigured. One such case arises when applying the classic equation that associates the current and voltage across a capacitor:

$$I(t) = C\frac{dV(t)}{dt}$$

This equation can be discretized in time and implemented as a simple dataflow model, shown in Figure 3.7. It is common in models of physical systems, such as Micro Electro-Mechanical Systems (MEMS) to have circuits with variable capacitances. Although it may seem reasonable to represent such a circuit by reconfiguring the capacitance parameter in the model, the original equation is not valid for variable capacitances. It is derived from the definition of capacitance, which is the ratio of charge to voltage:

$$C = \frac{Q}{V}$$

Rearranging and differentiating both sides gives the correct equation for describing a variable capacitor:

$$I(t) = \frac{dQ(t)}{dt} = C(t)\frac{dV(t)}{dt} + V(t)\frac{dC(t)}{dt}$$

This equation reduces to the earlier equation, as long as the capacitance does not change. Without explicitly representing the assumption that the capacitance parameter is constant, the model can be easily misused and the results misinterpreted.

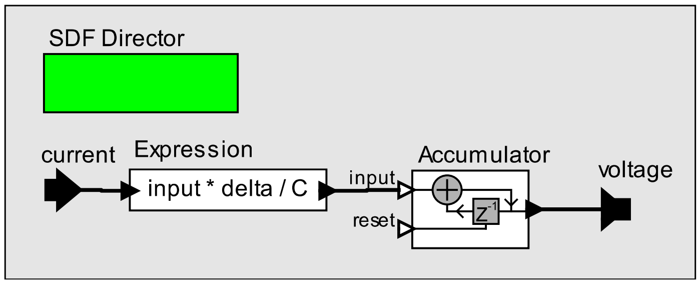


Figure 3.7: An example of a model with implicit assumptions that parameters do not change. In this model the capacitance parameter C should not be reconfigured, since the model was created assuming that the capacitance was constant. The parameter delta gives the amount of time between input samples of the current.

3.4.4 Reconfiguration and Dataflow Scheduling

As mentioned previously, scheduling analysis of synchronous dataflow models assumes that the token rates of ports do not change. However, model reconfiguration has the potential to affect token rates, although in many cases it does not. Other scheduling models allow token rates to change, as long as the changes occur only at certain points in the execution of a model. Ensuring that a model satisfies these constraints is a critical part of validating the correctness of models.

Unfortunately, understanding the dependencies between parameters that can be reconfigured and token rates is often difficult. As an example, Figure 3.8 shows an actor that describes a finite-impulse response (FIR) filter capable of decimation and interpolation. With the default parameter values shown, the actor performs no decimation or interpolation and produces and consumes a single token. However, if the decimation or interpolation parameters are given other values, then the actor will produce or consume multiple tokens each firing. For given interpolation and decimation factors, synchronous dataflow scheduling can be performed, but those parameter values must not be reconfigured. On the other hand, the taps parameter can be reconfigured without affecting dataflow scheduling.

In most dataflow modeling systems, the relationship between scheduling and reconfiguration is solved by design. Most systems provide a combination of scheduling algorithm, reconfiguration specification, and actor specification that is guaranteed to be safe. For

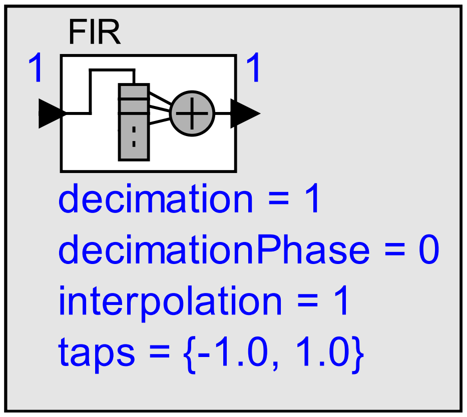


Figure 3.8: The FIR actor, showing the parameters that determine dataflow token rates. Because the decimation and interpolation parameters affect token rates, reconfiguration of this actor has the potential to violate scheduling assumptions.

instance, parameterized synchronous dataflow combines parameterized dataflow scheduling with reconfiguration ports and heterochronous dataflow combines modal models with delayed reconfiguration and on-the-fly synchronous dataflow scheduling. In these cases, assumptions about reconfiguration can remain implicit, since they are guaranteed by the structure of models that are allowed. Unfortunately, in a dataflow modeling system that supports hierarchically heterogeneous models with multiple scheduling algorithms and more that one way of specifying reconfiguration, the situation is more complex. Chapter 4 will present a unified formal framework aimed at explicitly specifying reconfiguration assumptions and verifying that these assumptions are satisfied in the presence of multiple sources of reconfiguration.

Chapter 4

Reasoning About Reconfiguration

This chapter presents an abstract, unified formalization of parameterization and reconfiguration in actor-oriented models. This formalization is abstract in the sense that it allows reconfiguration at all levels of the hierarchy, without binding reconfiguration to specific syntactic constructs. It represents static schedules, quasi-static schedules, token rates, type parameters, structural parameters, and user-level configuration options in a unified fashion. Dependencies between parameters are made explicit and may arise from a variety of sources, such as an expression in a design environment that expresses the value of parameter in terms of another, a declaration of token rates in a library actor, or a scheduler that synthesizes a schedule and corresponding token rates for the external ports of a model. Since it can be used to decide safety properties concerning the use of reconfiguration in a model, it is useful to think of this as a *behavioral type theory for reconfiguration*.

4.1 Parameterization Model

A *hierarchical reconfiguration model* is represented by a finite tree of actors, called the *containment tree*. Leaf elements of the tree are primitive, or *atomic* actors, and non-leaf elements are called *composite* actors. The root of the containment tree is the *toplevel* composite actor. The behavior of a composite actor is given by a dataflow model consisting of the actors that are its direct children in the tree. The dataflow model associated with each composite actor is assumed to reference *external ports* that communicate with the dataflow model that contains the composite actor. The composite actor at the root of the containment tree contains no external ports. We say that the all actors in a subtree are *contained* by the root of the subtree. Similarly, a composite actor *contains* all actors in the subtree rooted by the composite actor, including itself.

Formally, the set of actors in a model is $\mathbf{A}$. The parent of an actor is given by a partial function $parent : \mathbf{A} \rightarrow \mathbf{A}$, which is defined for all actors that are not toplevel composite actors. Recall that we can also consider *parent* to be a relation on actors, where $parent \subseteq \mathbf{A} \times \mathbf{A}$. By construction, *parent* is required to be irreflexive, corresponding to the constraint that no actor is its own parent. Furthermore, the reflexive, transitive closure $parent^*$ is required to be antisymmetric, corresponding to a constraint that no parent of an actor is also contained by the actor. $parent^*$ can also be interpreted as a partial order on actors, which will be called the *containment order* written it $\trianglelefteq$ to emphasize the fact that it is a partial order. The set of actors combined with the containment order is a tree, $(\mathbf{A}, \trianglerighteq)$, called the *containment tree*.

The set of parameters in a model is $\mathbf{P}$. Each parameter is associated with a single actor, given by the function $actor(p)$. For convenience, the subset of parameters associated with an actor a is written P_a. The value of each parameter at any point during execution of a model is given by an element of the set $\mathbf{V}$ of token values. In practical models, the values of parameters are often dependent on one another. This dependence might be specified explicitly in the construction of a model, e.g., one parameter is given as an expression of another, or implicitly, e.g., a dataflow scheduler synthesizes some parameter values. However, these differences are largely unimportant from the point of view of describing the constraints that parameter values must satisfy.

A *valuation function* is a function in $\mathbf{P} \rightarrow \mathbf{V}$ that gives the value of each parameter in a model. The value of a parameter p may depend on a finite, indexed set of parameters $domain^p = \{domain^p_1, \ldots, domain^p_n\}$. We say that a parameter p is *independent* if $domain^p$ is empty, and *dependent* otherwise. The value of each dependent parameter p is constrained by a *constraint function* $constraint_p : \mathbf{V}^n \rightarrow \mathbf{V}$, where n is the number of elements in $domain^p$. A *consistent valuation function* is a valuation function where the value of every dependent parameter satisfies the parameter's constraint function.

Definition 1 Consistent valuation function:

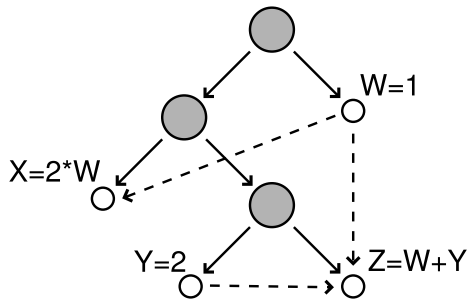


Figure 4.1: An example of parameters in a model. Actors are shaded nodes, parameters are unshaded. Solid lines depict the *parent* and *actor* relations, and dotted lines depict the $\rightsquigarrow$ relation.

A valuation function v is consistent if and only if $\forall p \in \mathbf{P}$, p is dependent $\implies \mathit{constraint}_p(v(\mathit{domain}^p_1), \ldots, v(\mathit{domain}^p_n)) = v(p)$

In a model, independent parameters in a model are allowed to be modified during reconfiguration, while dependent parameters cannot be. As a result, as long as the hierarchical structure of a model is fixed, the dependencies between parameters are fixed and can be statically analyzed. This model is essentially a version of an *attribute grammar* [25, 58] with fixed, finite structure.

Conceptually, if a parameter p is reconfigured, then all of the parameters that depend on it must be re-evaluated, followed by any parameter that depends on any of those, etc. The *dependence relation*, written $\rightsquigarrow \subseteq \mathbf{P} \times \mathbf{P}$, captures these dependencies. The dependence relation is the least transitive relation between parameters, such that $\forall x \in \mathit{domain}^p, x \rightsquigarrow p$. In order for a model to be well-defined, the dependence relation is required to be irreflexive, i.e., no parameter depends on itself. The set of parameters that are transitively modified by a parameter p will be written $\overset{\rightsquigarrow}{p} = \{x \in \mathbf{P}, p \rightsquigarrow x\}$ and, for a set of parameters P, $\overset{\rightsquigarrow}{P} = \bigcup_{p \in P} \overset{\rightsquigarrow}{p}$. Generally speaking, a design tool will determine the values of parameters in the set $\overset{\rightsquigarrow}{p}$ based on the value of a parameter p. Figure 4.1 shows the structure of the model graphically.

4.2 Reconfiguration Semantics

Formally, we write the set of all quiescent points of actor a during an execution of a model as Q^a. Hierarchical reactivity requires that $c \trianglerighteq a \implies Q^c \subseteq Q^a$. The set $\mathbf{Q} = \bigcup_{a \in \mathbf{A}} Q^a$ is the set of all quiescent points of all actors. The *precedence relation* is a partial order $\leq \subseteq \mathbf{Q} \times \mathbf{Q}$ that gives a time-ordering of quiescent points. The precedence relation is constrained such that the quiescent points Q^a of an actor a are totally ordered by $\leq$. If $q_1 \leq q_2$ then the quiescent point q_1 always occurs before q_2. If $q_1 \nleq q_2$ and $q_2 \nleq q_1$ then there is freedom in the execution of q_1 and q_2, possibly allowing for concurrent execution.

At each quiescent point q in the execution of a model, a set of independent parameters $R(q)$ is selected for reconfiguration. Based on this initial set of parameters and reconfigured values, reconfigured values for dependent parameters in $\overset{\rightsquigarrow}{R}(q)$ are determined based on their individual constraint functions and those parameters are also reconfigured. In general, this set of parameters $\overset{\rightsquigarrow}{R}(q)$ may be associated with actors anywhere in the model.

Although the set $R(q)$ contains a complete description of reconfiguration at a particular quiescent point, considering each quiescent point individually is generally unnecessary. It is more interesting to consider aggregate properties of an entire execution. During what set of quiescent points is a parameter reconfigured? Is a particular parameter reconfigured at all? The following two definitions, related to the notion of a *constant parameter*, describe these properties for a particular execution. These properties can be seen as bounds on the set of quiescent points during which a parameter is reconfigured. Two additional theorems give intuition about constant parameters.

Definition 2 Constant parameter:
Parameter p is *constant* if and only if
$\forall a \in \mathbf{A}, \forall q \in Q^a, p \notin \overset{\rightsquigarrow}{R}(q)$.

Definition 3 Constant parameter over actor firings:
Parameter p is *constant over firings of actor c* if and only if
$\forall a \in \mathbf{A}, \forall q \in Q^a, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$.

Theorem 1 *p is constant implies p is constant over firings of any actor.*

Theorem 2 *p is constant over firings of c and $c \trianglerighteq a$ implies p is constant over firings of a.*

As described in Chapter 3, actor-oriented models are often accompanied by reconfiguration requirements that must be satisfied for the model to be correct. If these requirements are not satisfied, then intrinsic assumptions of the model are violated and unexpected or undefined behavior may result. In the same sense that a program is *type safe* if constraints on the usage of data types are satisfied, an actor-oriented model is said to be *reconfiguration safe* if all necessary requirements on the use of reconfiguration are satisfied. In large actor-oriented models with many sources of reconfiguration and complex parameter dependencies, compile-time checking of reconfiguration safety is an important form of system verification.

Definition 4 Reconfiguration Requirement:
A *reconfiguration requirement* in a model m is a statement of the form "p is constant," or "p is constant over firings of actor a," where p and a are in the model.

Definition 5 Reconfiguration Safe:
An execution of a model with a set of reconfiguration requirements S is *reconfiguration safe* if the execution satisfies each requirement in S.

It is straightforward to cast the informal assumptions about reconfiguration described in 3 as formal requirements. If p is a type parameter used for static data type checking, then p must be constant in order to guarantee type soundness. Similarly, parameters that determine the structure of the model, such as the number of replications of a single actor, and parameters used for synchronous dataflow scheduling must also be constant. The local synchrony constraint for parameterized synchronous dataflow scheduling requires that parameters influencing the execution schedule of a composite actor c are constant over firings of c.

4.3 Change Contexts

In general, it is undecidable to determine if a parameter is constant or constant over firings of an actor on any particular execution, since the set $\mathbf{Q}$ is infinite and $R(q)$ for $q \in \mathbf{Q}$ might depend on data given to a model only at runtime. As a result, it is possible to detect violations of reconfiguration requirements at runtime but impossible to statically ensure reconfiguration safety in bounded time. Fortunately, by simplifying the properties being checked the problem can be made more tractable. Instead of analyzing the reconfiguration in particular executions, we will concentrate on analyzing *possible* reconfigurations in all executions.

To begin with, we assume that an actor-oriented model determines a *reconfiguration set* $R^a \subseteq \mathbf{P}$ for every actor a. The set R^a is the smallest set that contains all independent parameters that may be modified when actor a is quiescent. During any execution of the model, $\forall a \in \mathbf{A}, \forall q \in Q^a, R(q) \subseteq R^a$, and $\overset{\rightsquigarrow}{R}(q) \subseteq \overset{\rightsquigarrow}{R}^a$. For convenience, we say that an actor a is a *change context* for all parameters in R^a, and that a parameter is inherently constant (or inherently constant over actor firings) if its change contexts satisfy certain constraints. Intuitively, a parameter is inherently constant (over actor firings) if it is guaranteed to be constant (over actor firings) during any execution of the model.

Definition 6 Change context:
An actor a is a change context of a parameter p, written $a \rightsquigarrow p$, if and only if $p \in \overset{\rightsquigarrow}{R}^a$.

Definition 7 Inherently constant parameter:
Parameter p is *inherently constant* if and only if
$\forall a \in \mathbf{A}, a \not\rightsquigarrow p$.

Definition 8 Inherently constant parameter over actor firings:
Parameter p is *inherently constant over firings of actor a* if and only if $\forall c \in \mathbf{A}, c \rightsquigarrow p \implies c \trianglerighteq a$.

Theorem 3 *p is inherently constant implies p is constant during any execution.*

Theorem 4 *p is inherently constant over firings of actor c implies p is constant over firings of actor c during any execution.*

Given the set R^a for each actor and the parameter dependencies $domain^p$, definitions 7 and 8 can be used to check if a parameter is inherently constant. The definitions leave room for several direct computational procedures. One approach, shown in Figure 4.2, computes

```
Given: $\mathbf{A}, \mathbf{P}, R^a, domain^p$

for $p \in \mathbf{P}$ do
    changeContexts($p$) = $\phi$
    for $a \in \mathbf{A}$ do
        if ($p \in R^a$)
          then changeContexts($p$) += $a$
        fi
    od
od
$done$ = FALSE;
while (!$done$) do
        $done$ = TRUE;
        for $p \in \mathbf{P}$ do
            for $p' \in domain^p$ do
                if (changeContexts($p$) $\not\subseteq$ changeContexts($p'$))
                  then changeContexts($p$) += changeContexts($p'$);  $done$ = FALSE;
                fi
            od
        od
od
```

Figure 4.2: An algorithm for computing the set of change contexts. This set can be used to check reconfiguration requirements.

the *set of change contexts* for every parameter p, where $changeContexts(p) = \{a \in \mathbf{A} : a \rightsquigarrow p\}$. This set is then used to directly check the set of constraints C.

Although the above direct procedure for checking reconfiguration safety is decidable, it is not necessarily the most efficient approach. The complexity of the direct computation could be reduced after careful analysis of the problem, since the dependencies between parameters are usually sparse and long dependence chains are uncommon. Unfortunately, simple modifications of the algorithm in Figure 4.2 are fundamentally limited by the need to compute sets, such as $\{a \in \mathbf{A} : a \rightsquigarrow p\}$. It is also possible, however, to check reconfiguration safety of a model indirectly. The indirect approach offers an efficient computational algorithm for checking reconfiguration safety without directly dealing with set computations. In addition, it offers an intuitive conceptualization of reconfiguration requirements as constraints that must be satisfied by a model.

4.4 The Least Change Context

This section presents an alternative formulation of reconfiguration that is better suited to reasoning about reconfiguration. In particular, it leads to an efficient algorithm for checking inherent reconfiguration safety. Underlying this alternative formulation is the realization that the set $\{a \in \mathbf{A} : a \rightsquigarrow p\}$ of all change contexts of a parameter p can be usefully approximated by the greatest lower bound of the set when checking reconfiguration safety. This approximation arises primarily because the hierarchical structure of quiescent points mirrors the hierarchical structure of a model.

The least change context is not computed in the set $\mathbf{A}$, but instead in an artificially constructed set $\mathbf{A}_\perp^\top$ that contains artificial elements $\perp$ and $\top$. The element $\perp$ is less than all other elements and guarantees that $\mathbf{A}_\perp^\top$ with the appropriate ordering is a lattice. As a result, the greatest lower bound of a set of actors always exists in the set $\mathbf{A}_\perp^\top$, even though it may not in the set $\mathbf{A}$. The element $\top$ serves to represent the greatest lower bound of an empty set of actors. The $\top$ element allows constant parameters with no change contexts to be distinguished from parameters that are constant over firings of the toplevel actor.

Formally, the set $\mathbf{A}_\perp^\top$ is defined to be $\mathbf{A} \cup \{\top, \perp\}$ where $\top$ and $\perp$ are artificial elements not in $\mathbf{A}$. The ordering relation $\trianglerighteq_\perp^\top \subseteq \mathbf{A}_\perp^\top \times \mathbf{A}_\perp^\top$ is defined to be the transitive, reflexive, antisymmetric ordering relation where $\forall a \in \mathbf{A}, \forall b \in \mathbf{A}, a \trianglerighteq b \iff a \trianglerighteq_\perp^\top b$ and $\forall a \in \mathbf{A}_\perp^\top, \top \trianglerighteq_\perp^\top a \trianglerighteq_\perp^\top \perp$. With this construction, $(\mathbf{A}_\perp^\top, \trianglerighteq_\perp^\top)$ is a *lattice* [23]. A basic property of a lattice is that every set of elements A in the lattice has a greatest lower bound in the lattice. An example of a resulting lattice is shown in Figure 4.3.

We define the function $\lfloor \cdot \rfloor : \mathbf{P} \rightarrow \mathbf{A}_\perp^\top$ as shown in Definition 9 and say that $\lfloor p \rfloor$ is the *least change context* of the parameter p. The least change context of a parameter p is essentially a conservative approximation of the set of all the change contexts of p. If the least change context of a parameter p is either $\top$ or an element of $\mathbf{A}$, then the set of change contexts of p is limited and reconfiguration of p can only occur during the quiescent points of certain actors. On the other hand, if the least change context is $\perp$ then the conservative approximation gives no interesting information about reconfiguration, and no restrictions on reconfiguration can be inferred. Theorems 5 and 6 prove the soundness of the least change context approximation.

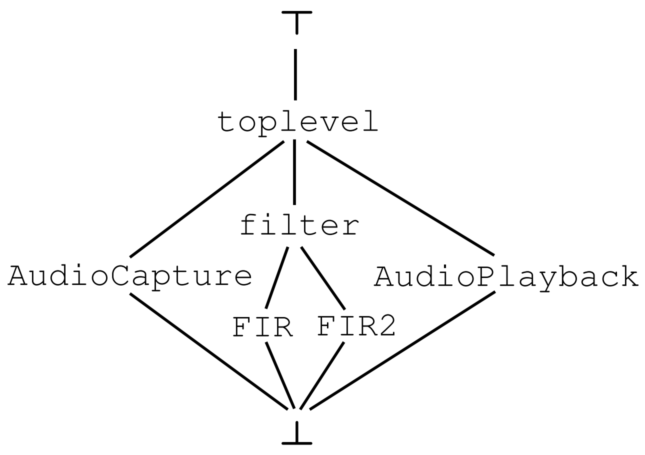


Figure 4.3: An example of the lattice formed by augmenting the containment tree of the model in Figure 2.2 with artificial top and bottom elements.

Definition 9 Least change context of a parameter:
The least change context of a parameter p, $\lfloor p \rfloor$, is an element of $\mathbf{A}_{\perp}^{\top}$ where $\lfloor p \rfloor = \sqcap\{a \in \mathbf{A}_{\perp}^{\top} : a \in \mathbf{A} \wedge a \leadsto p\}$
Or equivalently,

$$\lfloor p \rfloor = \begin{cases} \top & \text{if } \{a \in \mathbf{A} : a \leadsto p\} = \emptyset \\ \sqcap\{a \in \mathbf{A} : a \leadsto p\} & \text{if } \{a \in \mathbf{A} : a \leadsto p\} \neq \emptyset \text{ and } \\ & \sqcap\{a \in \mathbf{A} : a \leadsto p\} \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Theorem 5 $\lfloor p \rfloor = \top$ *implies p is inherently constant.*

Theorem 6 $\lfloor p \rfloor \in \mathbf{A}$ *implies p is inherently constant over firings of* $\lfloor p \rfloor$.

Approximate approaches to static analysis must always balance usefulness with utility and avoid discarding interesting information about behavior. One source of approximation in our theory arises from the inherently constant property, which requires that reconfiguration of a parameter not occur during *any* behavior of the model. While it is possible to construct models that specify reconfiguration that does not actually occur, such as a modal model where the guards of transitions are always false, we accept that such models might be rejected by by reconfiguration analysis. A second source of approximation arises from the least change context approximation to the set of change contexts. Theorem 7 shows that for interesting reconfiguration requirements, such as the local synchrony constraint for parameterized synchronous dataflow scheduling, the least change context approximation does not discard information.

Theorem 7 *p is inherently constant over $actor(p)$ implies that $\lfloor p \rfloor \neq \bot$.*

Based on the structure of a model, the least change context of a parameter must satisfy two constraints over the lattice $(\mathbf{A}_\bot^\top, \trianglerighteq_\bot^\top)$. The first constraint (Theorem 8) requires that the least change context of a parameter p cannot be any higher in the hierarchy than the least change context of a parameter that p depends on. The second constraint (Theorem 9) requires that if a parameter is reconfigured by an actor, then the actor must contain the least change context of the parameter. In fact, these constraints will be satisfied by not only the least change context but also *any* lower bound on the set of change contexts. Using the greatest lower bound, however, gives the most information about the set of change contexts for a parameter.

Theorem 8 *$p_1 \rightsquigarrow p_2$ implies $\lfloor p_1 \rfloor \trianglerighteq_\bot^\top \lfloor p_2 \rfloor$.*

Theorem 9 *$p \in R^c$ implies $c \trianglerighteq_\bot^\top \lfloor p \rfloor$.*

By using the above constraints, the least change context of a parameter can be computed *without direct computation of the set of change contexts for each parameter*. One algorithm for computing the solution is known to be linear time in the number of constraints [87]. The algorithm computes $\lfloor \cdot \rfloor$ by beginning with an initial guess where $\forall p \in \mathbf{P}, \lfloor p \rfloor = \top$. The initial guess is updated according to each constraint until all the constraints are satisfied.

4.5 Conditional Reconfiguration

Up to this point, we have considered unconditional specifications of reconfiguration. However, in some cases reconfiguration is useful to consider reconfiguration to be *conditional*. Depending on the structure of the model, or the values of other parameters, reconfiguration might or might not actually occur. If reconfiguration does not occur, then this fact is often important to expose to the behavioral type system in order to prevent models

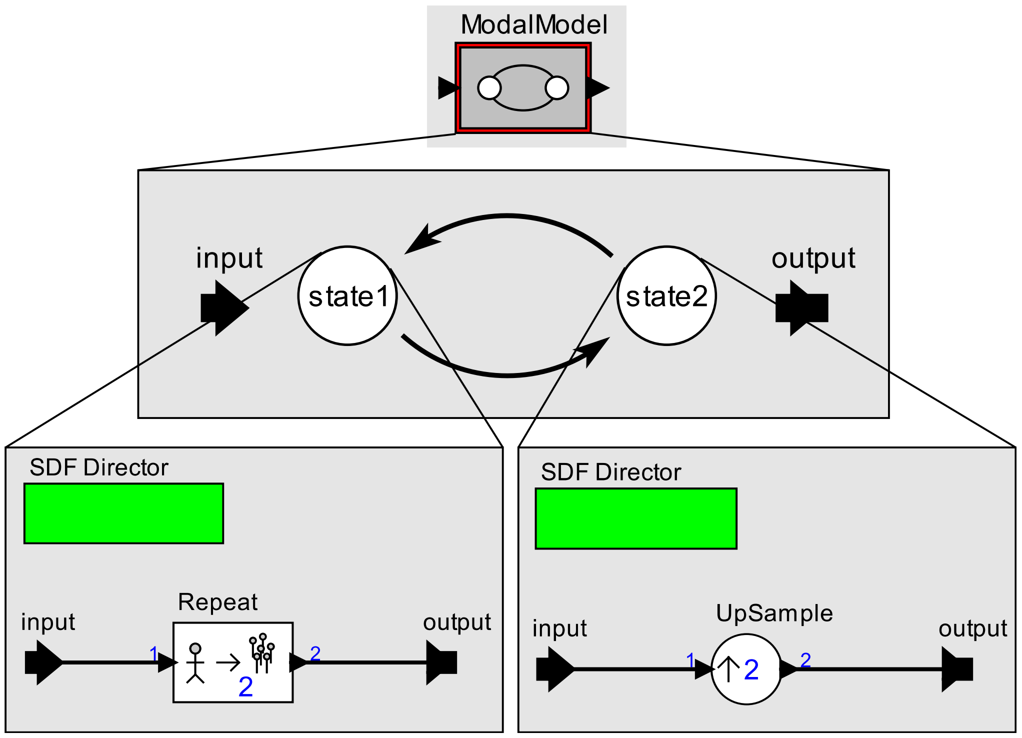


Figure 4.4: An example of a component that exhibits conditional reconfiguration. Every time the modal model makes a state transition, the external rates are reconfigured to the same value.

that are reconfiguration safe, but not inherently reconfiguration safe, from being considered invalid. As an example, the model in Figure 4.4 shows one case in which it is useful to consider conditional reconfiguration. This model is similar to the model in Figure 3.3, except that each refinement operates on the same number of tokens.

Can this component can be used in an SDF model? Since each state refinement has the same input and output rates, one might think that it should be possible. However, the formal framework above provides no way of considering this information in the set R^c and must conservatively assume that the rate parameters of the modal model are reconfigured. Another example is shown in Figure 4.5. It seems logical to consider the rate parameter of the actor to be constant, since the reconfiguration port always receives the same value. However, the formal framework again provides no mechanism for considering this information.

Conditional reconfiguration also arise simply through parameter dependencies. If a

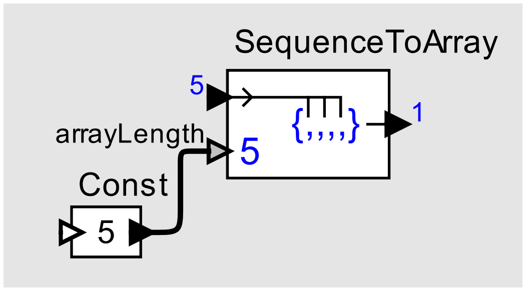


Figure 4.5: Another example of a partial model that exhibits conditional reconfiguration. In this model, the value received from the bottom port changes the number of tokens consumed by the top port. If the parameter of the Const actor is not reconfigured, then the rate parameter of the input port obviously doesn't change, even though it is reconfigured.

parameter a depends on parameters x and y, then we assume that whenever x or y changes, then a changes. However, if the constraint function for a is

$$constraint_a(x, y) = \begin{cases} 0 & \text{if } x < 0 \\ y & \text{otherwise} \end{cases}$$

and x is less than 0 and never reconfigured, then a does not change when y changes. The constraint in Theorem 8 is safe, but *conservative*.

In order to accommodate conditional reconfiguration, the left-hand side of the constraints in Theorems 8 and 9 may be augmented to include a *conditional function* f. A conditional function takes the initial parameter valuation v_0, and the least change context function $\lfloor \cdot \rfloor$ and returns a new lower bound on the least change context. The updated constraints are shown below, in terms of Definitions 10 and 11. The resulting solution to the least change context can be used to check if parameters are constant, even if they are not inherently constant.

Definition 10 Conditional Reconfiguration Function:
A *conditional reconfiguration function* $f_{a \rightsquigarrow p} : (\mathbf{P} \rightarrow \mathbf{V}) \times (\mathbf{P} \rightarrow \mathbf{A}_{\bot}^{\top}) \rightarrow \{a, \top\}$ for an actor a and a parameter p is monotonic function, where $f^a(v_0, \lfloor \cdot \rfloor) = a$ if in any execution beginning with parameter values v_0, $\forall q \in Q^a, p \in R(q)$.

Definition 11 Conditional Dependence Function:
A *conditional dependence function* $f_{p_1 \rightsquigarrow p_2} : (\mathbf{P} \to \mathbf{V}) \times (\mathbf{P} \to \mathbf{A}_{\perp}^{\top}) \to \mathbf{A}_{\perp}^{\top}$ for parameters p_1 and p_2 is monotonic function, where $f_{p_1 \rightsquigarrow p_2}(v_0, \lfloor \cdot \rfloor) = \lfloor p_1 \rfloor$ if in any execution beginning with parameter values v_0, reconfiguration of p_1 requires evaluation of $constraint_{p_2}$

$p \in R^c$ implies $f_{a \rightsquigarrow p}(v_0, \lfloor \cdot \rfloor) \trianglerighteq_{\perp}^{\top} \lfloor p \rfloor$
$p_1 \rightsquigarrow p_2$ implies $f_{p_1 \rightsquigarrow p_2}(v_0, \lfloor \cdot \rfloor) \trianglerighteq_{\perp}^{\top} \lfloor p_2 \rfloor$

The modified constraints are somewhat more complicated, since the least change context appears in a function on the left hand side of the inequality. As a result, if f is not well behaved then the least change context may no longer be well defined or easily determined computationally. To ensure that the algorithm in [87] operates, it is sufficient to require that f is a *monotonic function*, i.e., if l_1 and l_2 are functions in $\mathbf{P} \to \mathbf{A}_{\perp}^{\top}$, then $\forall p \in \mathbf{P}, l_1(p) \trianglerighteq_{\perp}^{\top} l_2(p)$ implies $f(v_0, l_1) \trianglerighteq_{\perp}^{\top} f(v_0, l_2)$.

In general, it is important to notice that conditional reconfiguration can be difficult to detect statically. For instance, it is generally undecidable whether a stream in a dataflow model has a constant sequence of values, or not. However, the formal framework can be extended to support any analysis to detect conditional reconfiguration, by incorporating more model semantics into f^a. Even fundamentally undecidable techniques such as model checking could be used, although we expect that most design tools will follow a type-based philosophy and accept that static safety analysis will flag some safe models as errors. Ultimately, it is the responsibility of individual design tools to provide analysis which is appropriate to support a designers intuition of which models should be valid.

Chapter 5

Design Examples

This chapter presents some significant designs in Ptolemy II which illustrate the design issues that reconfiguration analysis addresses.

5.1 Blind Communication Receiver

Figure 5.1 shows an example signal processing model that describes a *blind communication receiver*. This system is designed to analyze and process a received signal with unknown characteristics to determine the carrier frequency, baud rate, and number of phase shifts of a digital Phase Shift Keyed (PSK) signal. The toplevel model is executed in the style of a Kahn-MacQueen process network [48, 69], where each actor is associated with an operating system thread and actor threads block until communication queues have enough data. Most of the actors in the process network are defined hierarchical using statically scheduled dataflow models, resulting in a hierarchically heterogeneous composition.

The Demodulator and BaudRateEstimator actors are implemented by synchronous dataflow models that process 2^{order} input samples and compute estimates of the carrier frequency and symbol rate of the input signal. Additionally, the Demodulator block synthesizes a carrier signal of the appropriate frequency and outputs a baseband version of the input signal. The Resampler actor samples the baseband signal at the estimated baud rate and outputs a data-dependent number of complex samples. The PhaseStatesEstimator processes the resampled data to estimate the number of different phases used in the PSK transmission.

A hierarchical model implementing the PhaseStatesEstimator using a dynamically scheduled dataflow model is shown in detail in Figure 5.1. This model relies on the ComputeHistogram actor, which computes an array representing a histogram of input data. The number of samples used to compute the histogram is specified as an actor parameter reconfigured by the inputCount reconfiguration port. The model is constructed so that a histogram is computed of all the resampled data.

Overall, the data-dependent nature of the resampling operation prevents the toplevel model from being statically scheduled, since the number of resampled data tokens is not available to a scheduler. However, in order to avoid the overhead of runtime scheduling, a static or quasi-static schedule for the PhaseStatesEstimator would be preferred. Attempts to apply synchronous dataflow scheduling analysis to the model results in constraints that cannot be satisfied, since the parameter that determines the number of tokens consumed by the histogram is reconfigured. Direct application of parameterized synchronous dataflow scheduling to the model also fails, since the PhaseStatesEstimator actor is not locally synchronous. The inconsistent constraints derived from sources of reconfiguration, parameter dependencies, and reconfiguration requirements for such models are shown in Figure 5.2.

One design solution is to modify the model as shown in Figure 5.3. In this model, reconfiguration has been moved up one level in the model, resulting in reconfiguration just before the PhaseStatesEstimator is fired. The value of the inputCount parameter is equal to value of the count parameter, which is reconfigured by a reconfiguration port. In this model, the PhaseStatesEstimator model is locally synchronous, as indicated by the reconfiguration constraints in Figure 5.4. The resulting execution schedule is quasi-static and depends on the current rate parameter of the ComputeHistogram actor, which is reconfigured by the count reconfiguration port.

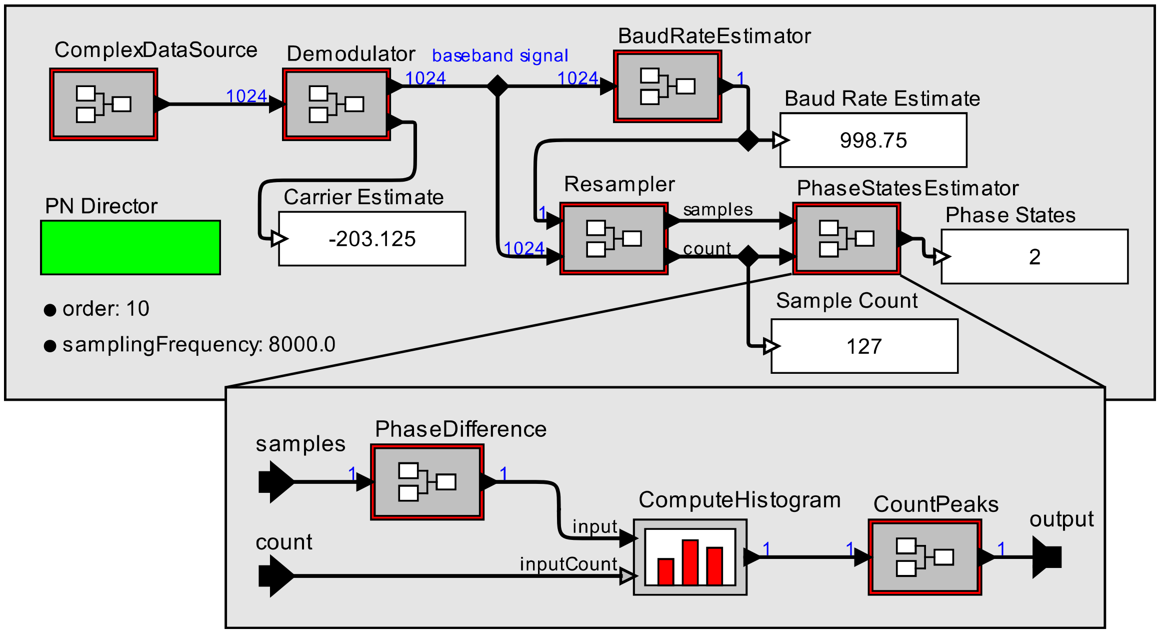


Figure 5.1: A process network design example where each actor is an independent thread that blocks waiting for input data.

$$
\begin{aligned}
\text{ComputeHistogram} \quad & \trianglerighteq^{\top}_{\bot} \quad \lfloor \text{ComputeHistogram.inputCount} \rfloor \\
& \trianglerighteq^{\top}_{\bot} \quad \lfloor \text{ComputeHistogram.input.tokenConsumptionRate} \rfloor \\
& \trianglerighteq^{\top}_{\bot} \quad \top
\end{aligned}
$$

(a) Inconsistent SDF Scheduling Constraints

$$
\begin{aligned}
\text{ComputeHistogram} \quad & \trianglerighteq^{\top}_{\bot} \quad \lfloor \text{ComputeHistogram.inputCount} \rfloor \\
& \trianglerighteq^{\top}_{\bot} \quad \lfloor \text{ComputeHistogram.input.tokenConsumptionRate} \rfloor \\
& \trianglerighteq^{\top}_{\bot} \quad \lfloor \text{ComputeHistogram.PSDFschedule} \rfloor \\
& \trianglerighteq^{\top}_{\bot} \quad \text{PhaseStateEstimator}
\end{aligned}
$$

(b) Inconsistent PSDF Scheduling Constraints

Figure 5.2: Reconfiguration constraints associated with different types of dataflow scheduling analysis of the model in Figure 5.1. The constraints in (a) indicate that the model is not a valid SDF model. The constraints in (b) indicate that the model is not locally synchronous, and hence not a valid PSDF model either. In each case, the final constraint corresponds to a reconfiguration requirement that is violated.

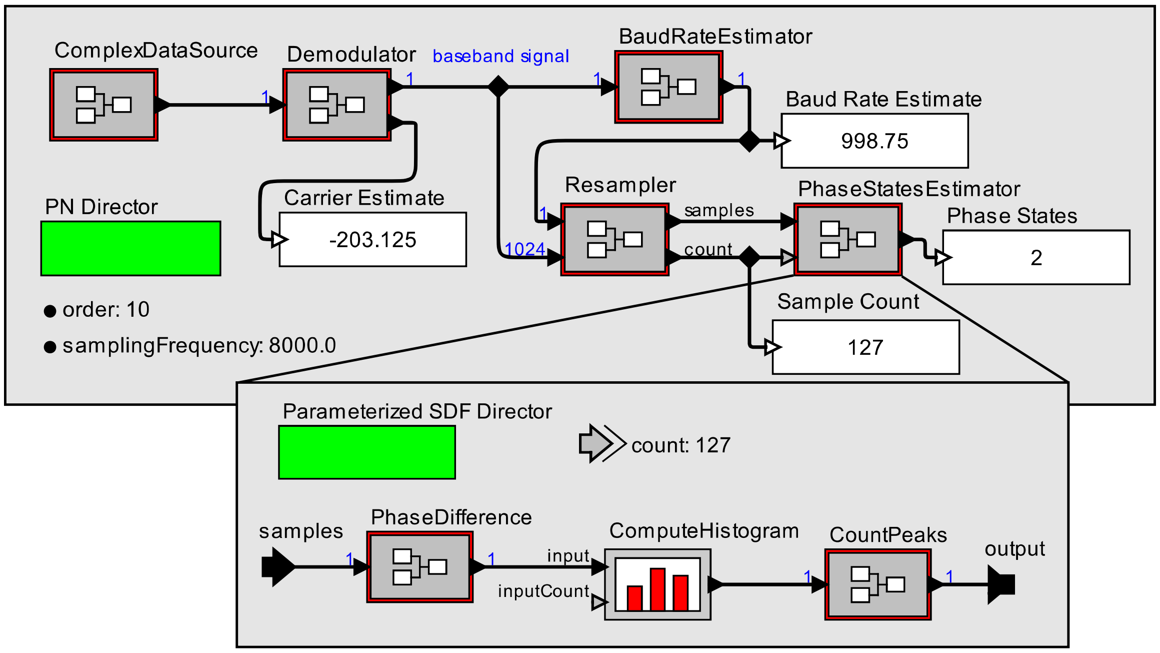


Figure 5.3: An improved design that allows more opportunities for static dataflow scheduling. The `count` port has been converted from a dataflow port to a reconfiguration port, and the PhaseStatesEstimator model has been changed to use a parameterized synchronous dataflow scheduler.

$$
\begin{aligned}
\text{PhaseStateEstimator} \quad & \trianglerighteq^{\top}_{\perp} \quad \lfloor \text{ComputeHistogram.inputCount} \rfloor \\
& \trianglerighteq^{\top}_{\perp} \quad \lfloor \text{ComputeHistogram.input.tokenConsumptionRate} \rfloor \\
& \trianglerighteq^{\top}_{\perp} \quad \lfloor \text{ComputeHistogram.PSDFschedule} \rfloor \\
& \trianglerighteq^{\top}_{\perp} \quad \text{PhaseStateEstimator}
\end{aligned}
$$

Figure 5.4: Consistent reconfiguration constraints associated with the model in Figure 5.3. Again, the final constraint is implied by a reconfiguration requirement that must be satisfied. These constraints, along with others not shown, indicate that the model is locally synchronous and can be scheduled using parameter synchronous dataflow techniques.

5.2 Rijndael Encryption

A second example of reconfiguration is shown in Figure 5.5, which performs the AES-standard Rijndael encryption algorithm [22]. This algorithm performs a sequence of 10 encryption rounds on blocks of 16 bytes. In this model, each block of data is communicated in sequence between each actor. The encryption operations are all performed by the RoundSequence component, where data is fed back to the input as needed for future encryption rounds. The RoundKeyGenerator generates a pseudo-random sequence from the user's 16-byte key, an operation called *key expansion*. Since each encryption round uses a fresh portion of the pseudo-random sequence generated by RoundKeyGenerator, this architecture makes distribution of round keys relatively simple.

At the top level, the Rijndael model operates on blocks of 16 bytes at a time. The sequence of rounds is governed by the RoundSequence modal model. In the first round, the incoming block of tokens is read and XOR'd with round key. The resulting 16 bytes are produced on the intermediate cipher port and fed back to the modal model. The main encryption rounds operate entirely on intermediate cipher values returned to the last cipher port, and no further tokens are read from the text input. In the last encryption round, 16 bytes representing the final encrypted text is produced on the cipher port, completing the encryption process. The Sbox, ShiftRow, and MixColumn actors implement the corresponding operations in the Rijndael specification.

Given understanding of the model, it is not too hard in this case to determine the dataflow behavior of the Rijndael actor. Using robust run-time scheduling of dynamic dataflow graphs [7, 69, 83], the operations do not deadlock and execute forever in a bounded amount of memory for communication. However, this fact cannot be proven using either synchronous dataflow or parameterized synchronous dataflow analysis, since the rate parameters of the RoundSequence model are reconfigured during each state transition. The fact that these assumptions are not satisfied is indicated by inconsistent reconfiguration constraints, as shown in Figure 5.2.

One approach to recovering the robustness of static scheduling is to modify the model to include "dummy" communication that makes data rates constant, as shown in Figure 5.6. In this model, every state of the RoundSequence actor produces and consumes 16

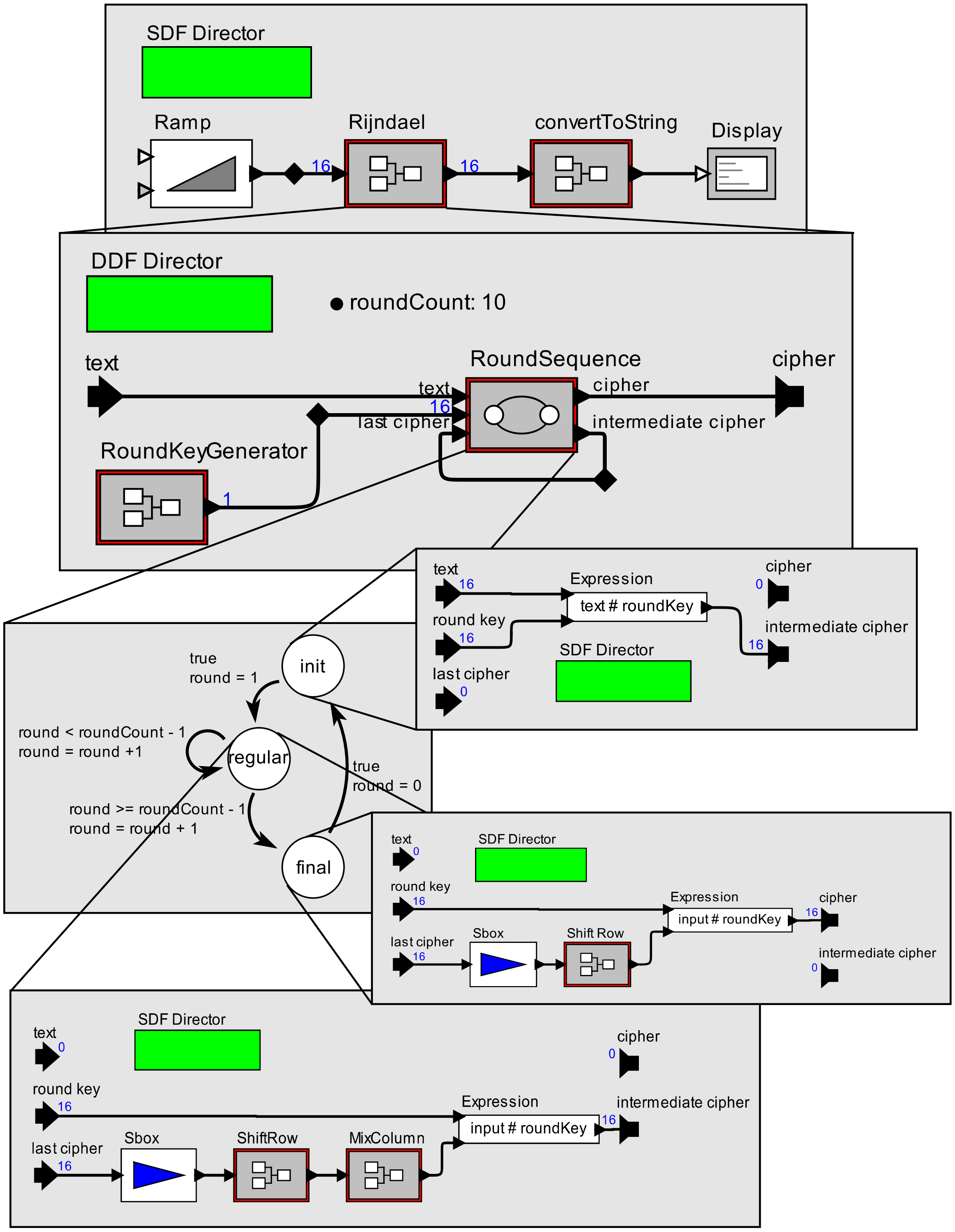


Figure 5.5: A model of the Rijndael Encryption algorithm.

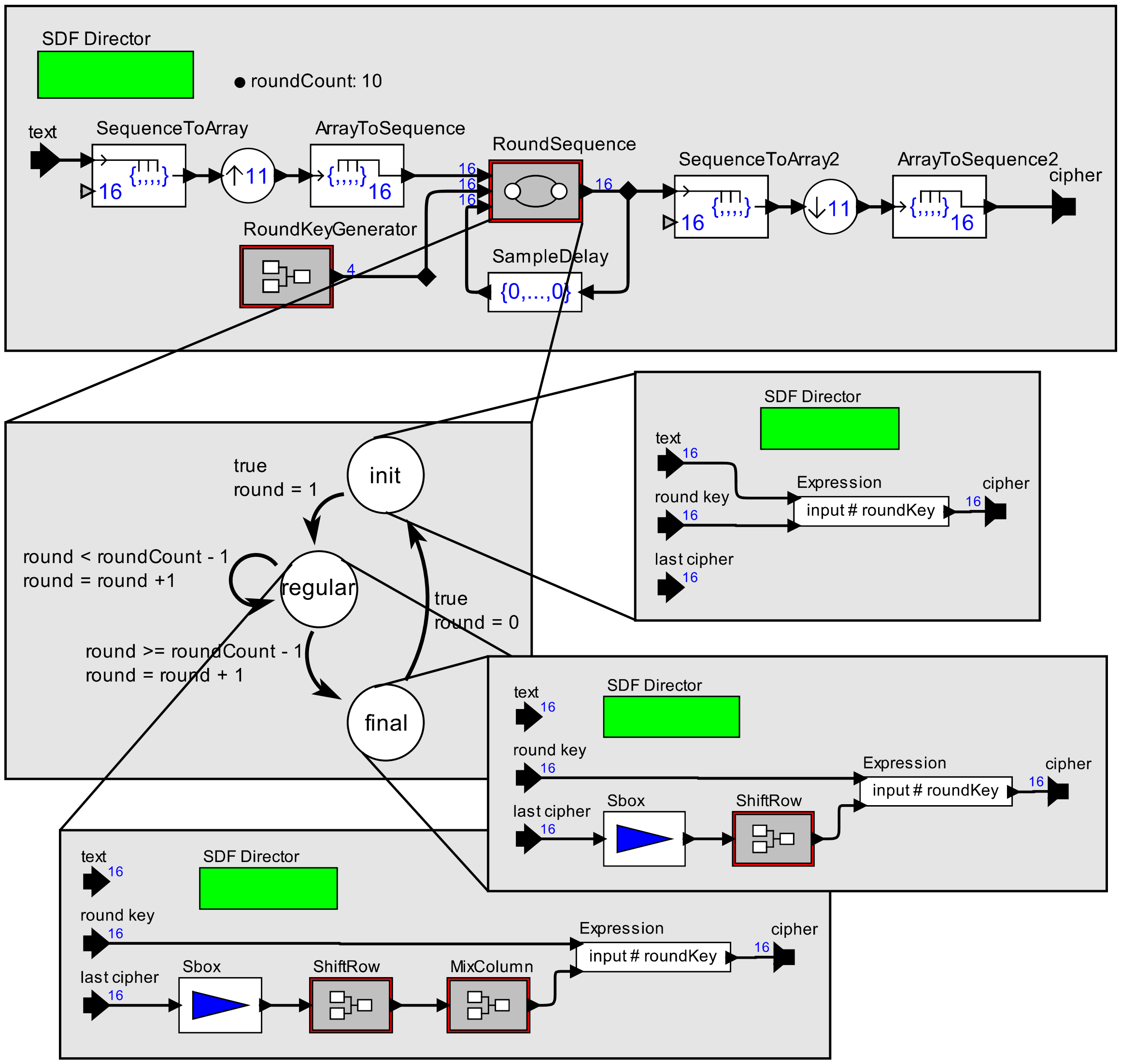


Figure 5.6: A synchronous dataflow model of the Rijndael Encryption algorithm.

bytes of data in sequence. Unneeded data is read and discarded. The external behavior of the model is identical to the model in Figure 5.5 due to "dummy" data produced by upsampling the incoming data and by the SampleDelay. Intermediate outputs are discarded by downsampling the output to leave just the final result.

In this model, the reconfiguration constraints of synchronous dataflow scheduling can be guaranteed, as shown in Figure 5.7. Note that these constraints use a conditional reconfiguration function to assert that rate parameters of the modal model are not reconfigured

$$f_{\mathsf{RoundSequence} \rightsquigarrow \mathsf{tokenConsumptionRate}}(v_0, \lfloor\cdot\rfloor) =$$

$$\begin{cases} \top & \text{if} \quad \begin{array}{c} v_0(\mathsf{init.text.tokenConsumptionRate}) = \\ v_0(\mathsf{regular.text.tokenConsumptionRate}) = \\ v_0(\mathsf{final.text.tokenConsumptionRate}) \\ \text{and} \\ \lfloor\mathsf{init.text.tokenConsumptionRate}\rfloor = \\ \lfloor\mathsf{regular.text.tokenConsumptionRate}\rfloor = \\ \lfloor\mathsf{final.text.tokenConsumptionRate}\rfloor = \top \end{array} \\ \mathsf{RoundSequence} & \text{otherwise} \end{cases}$$

$$\begin{aligned} f_{\mathsf{RoundSequence} \rightsquigarrow \mathsf{tokenConsumptionRate}}(v_0, \lfloor\cdot\rfloor) &\trianglerighteq_{\bot}^{\top} \lfloor\mathsf{RoundSequence.text.tokenConsumptionRate}\rfloor \\ &\trianglerighteq_{\bot}^{\top} \lfloor\mathsf{Rijndael.SDFschedule}\rfloor \\ &\trianglerighteq_{\bot}^{\top} \top \end{aligned}$$

Figure 5.7: Reconfiguration constraints associated with synchronous dataflow scheduling analysis of the model in Figure 5.6. The constraints indicate that the model can be scheduled using synchronous dataflow techniques, as long as the rate parameters of the modal model are conditionally reconfigured.

if the initial values of the refinement rate parameter in each mode are equal and inherently constant. Execution of the resulting static schedule is guaranteed to execute forever in bounded memory without deadlock. The disadvantage of this model is that, depending on the desired implementation architecture, the dummy communication may result in undesirable overhead. Because of this overhead, it may be desirable to refactor the model further to leverage less constrained scheduling analysis, such as cyclo-static dataflow [12] or cyclo-dynamic dataflow [98].

Chapter 6

Actor-Oriented Metaprogramming System

This chapter describes a system for software design based on actor-oriented metaprogramming. The system consists of two primary portions: Ptolemy II and Copernicus. Ptolemy II is a design environment targeted primarily at capturing abstract system behavior using highly reusable and reconfigurable components and models. Although Ptolemy II models are directly executable, the inherent overhead in the framework is generally unacceptable for models constructed from fine-grained reusable components. Models of software systems are generally larger, consume more memory, and execute slower than optimized hand-written code.

Copernicus attempts to reduce this execution penalty by automatically eliminating generic aspects of components in a model through actor specialization. The process leverages significant portions of the functional descriptions of actors and data types in Ptolemy II through generalized partial-evaluation techniques. For portions of the framework where a wider variety of implementation possibilities are desired or where partial-evaluation techniques perform poorly, explicit specifications of generated code can be used instead. The resulting Java code is much closer to what a designer might write by hand, while still leveraging actor-oriented design techniques.

The presentation here is not intended to be a comprehensive description of Ptolemy II, or Copernicus. Instead, we will focus on how actor-oriented models are represented, with an emphasis on data, type, and component abstractions.

6.1 Ptolemy II

Ptolemy II [44] is a design tool supporting the actor-oriented design of systems. It is implemented as a Java class library that models actor-oriented syntax and semantics. Ptolemy II provides a variety of extension points for adding new actors, new data types, and new models of computation within the basic framework. The ptolemy.actor package is the basis for actor-oriented models in Ptolemy II.

The actor package implements an object-oriented framework for modeling actor-oriented systems. It includes structures for representing actors (the TypedAtomicActor class), ports and parameters (the TypedIOPort and Parameter classes), and for describing compositions of actors in a model (the TypedCompositeActor class). In a model, connections between ports are represented by relations (the TypedIORelation class), while the model of computation is determined by a director (the Director base class). The director is responsible for creating receivers (the Receiver interface) of the appropriate type to manage communication and for driving the execution of individual actors. Models are also associated with a single writer, multiple reader locking mechanism, implemented by the Workspace class. This locking mechanism allows the structure of models to be modified safely in a multi-threaded environment.

Ptolemy II supports a rich syntax for actor-oriented models. Ports may be *multiports* and connected to multiple relations. Relations may have an associated *width*, allowing them to represent multiple communication channels with a single connection. The number of independently addressable communication channels for a port is inferred from the number and width of relations connected to it. Syntactically, multiple ports can also be connected to a single relation, allowing *fan-out* or *merging* of communication channels when such structures are allowed in a particular model of computation.

Parameterization is represented in Ptolemy II by instances of the Parameter class. Parameters are associated with a string *expression*, which is evaluated to determine the *value* of the parameter, represented by a data token. Expressions may reference the values of other parameters, allowing parameter values to depend on one another. Expressions are

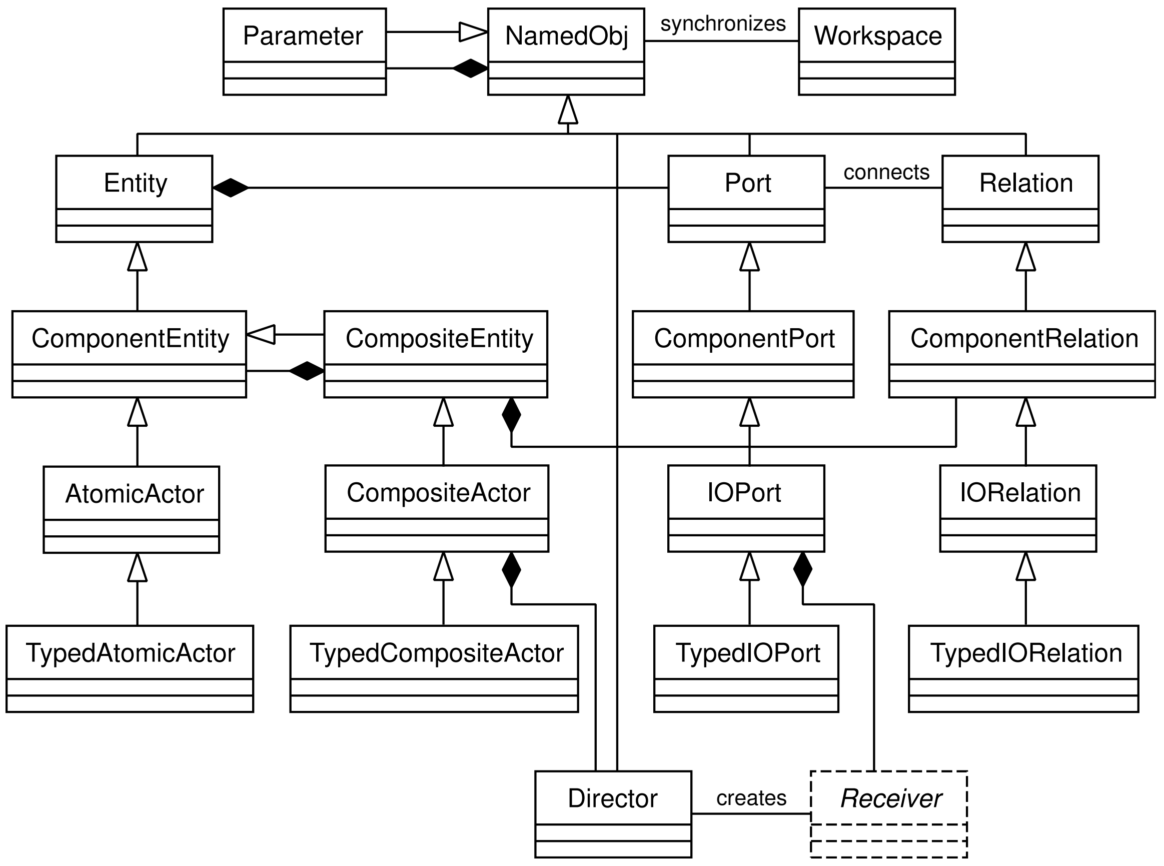


Figure 6.1: A UML diagram for Ptolemy II's ptolemy.actor package, showing various supporting classes

assumed to be *functional*, indicating that evaluation of a parameter expression may not change the state of the system. This property allows significant implementation freedom, since the value may either be cached or repeatedly recomputed when needed.

Ptolemy II emphasizes the construction of highly reusable actor specifications, which may be instantiated in a context and given parameter values and connections appropriate to that context. Fundamentally actors are components that can be generally connected and reconnected to other actors. Actors can be parameterized and reconfigured during execution, as described in Section 3. Additionally, actors in Ptolemy II are designed to be *type-polymorphic* (able to operate on data of different types) and *domain-polymorphic* (able to operate under different models of computation). Type polymorphism is largely supported by the ptolemy.data package, which hides arbitrary data objects behind an object-oriented

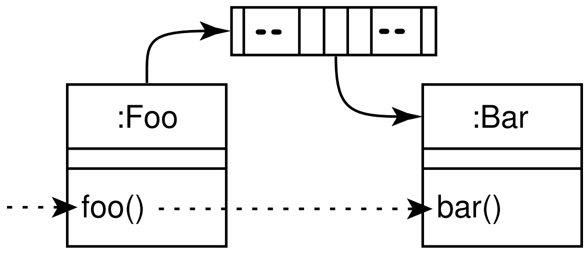


Figure 6.2: A simple indirection diagram showing invocation of the `bar()` method of an instance of the Bar class from the `foo()` method of an instance of the Foo class. The instance of the Bar class is obtained by indexing into the array referenced by the instance of the Foo class.

abstract data type. Domain polymorphism is supported by the Actor interface, which is implemented by all actors and directors. Parameters, type-polymorphism, connections, and domain-polymorphism are the primary *generic aspects* of actor-oriented design.

6.1.1 Indirection in Object-oriented Frameworks

As in typical object-oriented frameworks, Ptolemy II implements generic aspects of actors through interfaces and indirection. This indirection is not visible in the UML static structure diagrams of the previous page. The indirection in the implementation of each method will be described using an *indirection diagram*, an example of which is shown in Figure 6.2.

An indirection diagram has the general structure of a UML object diagram, where each box represents an instance of a Java class. An object with a dotted outline represents an instance of an unknown subclass of the class or interface. References between the objects are represented by a solid arrow between the objects, or by a solid line for references navigable in both directions. Methods are shown in the body of an object in a similar manner to a UML class diagram. Method invocations are represented by dotted arrows from one method to another, with multiplicities representing the possibility of multiple invocations. For convenience, entry point invocations are shown as an arrow with no source.

Indirection diagrams are similar in many ways to a UML collaboration diagram, since they describe method invocations between different objects. However, unlike a collaboration diagram which emphasizes the sequencing of method calls, an indirection diagram emphasizes the implementation of particular methods. An indirection diagram is primarily used in the examples below to show the approximate complexity of various methods in the Ptolemy II framework.

6.1.2 Data and Data Types

In order to maximize reuse of actors in a model, Ptolemy II provides base classes for representing data types in a uniform way through the ptolemy.data package. Various subclasses of the Token base class encapsulate both Java primitive types and composite types such as arrays and records. By programming using the Token class as an abstract data type, type polymorphic Java code can be written.

Data types are explicitly represented in Ptolemy II through the ptolemy.data.type package. Various subclasses of the Type base class represent classes of token values. For most Token subclasses encapsulating Java primitive types, such as the IntToken class that represents operations on signed integers, there is a corresponding Type subclass and a singleton instance of that subclass. For other tokens, such as the ArrayToken class, which encapsulates an array of other tokens, multiple instances of a single Type subclass are instantiated to represent different contained token types. Conceptually, data types represent subsets of the set $\mathbf{V}$ of all token values. Every value $v \in \mathbf{V}$ has a unique *exact type* given by $\mathit{Type}(v)$, where $\mathit{Type} : \mathbf{V} \rightarrow \mathbf{T}$. The set $\mathbf{V}_\tau = \{v \in \mathbf{V} : \mathit{Type}(v) = \tau\}$ is the set of all tokens with type τ.

Data types are related in a lattice, as shown in Figure 6.3, where automatic conversion is allowed from one type to another. This conversion happens during operations between tokens and during communication from one actor to another. If $\tau \leq \tau'$ in the type lattice then automatic type conversion is allowed from values with type τ to values with type τ'. The *automatic conversion function* $\mathrm{Convert}_\tau$ is a partial function where $\mathrm{Convert}_\tau : \mathbf{V} \rightarrow \mathbf{V}_\tau$ returns the result of converting an arbitrary value to a value of type τ. The conversion function satisfies two primary constraints: conversion to a lesser type is not possible, and conversion of a value of one type to the same type is an identity operation.

This lattice includes both exact types (shown unshaded) and *abstract* types (shown

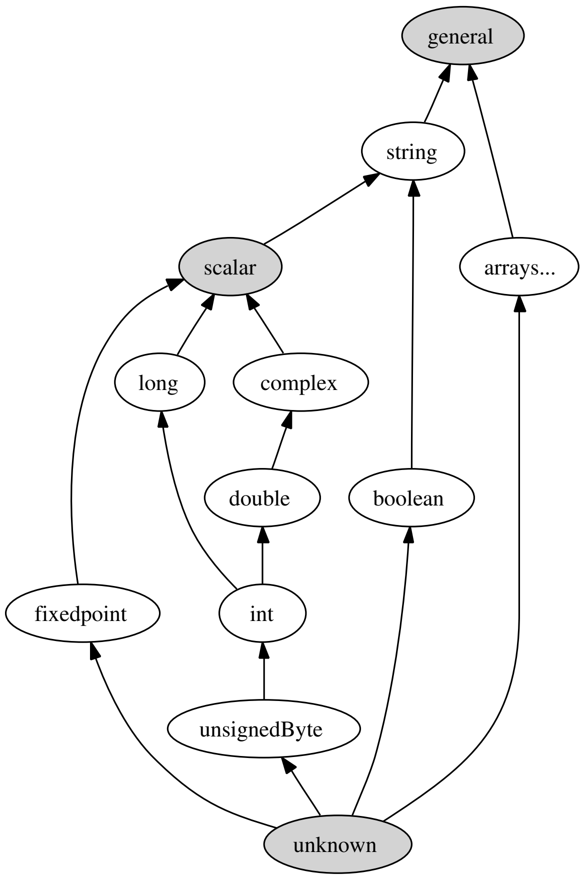


Figure 6.3: An abbreviated version of the Ptolemy II data type lattice.

shaded). Generally speaking, exact types represent disjoint sets of tokens and automatic conversion to an exact type results in a different token. On the other hand, abstract types represent the union of all lesser data types and conversion to an abstract type does not result in a new value. The type general corresponds to the Token base class and includes every token value. The type unknown is an artificial type that is only used in type checking.

Although the interface defined in the Token base class make it easy for a designer to describe type-polymorphic operations, there is additional overhead to support these operations as shown in Figure 6.4. Data types must be compared during each operation in order to ensure that operations are value and to allow for automatic type checking. In addition, in order to guarantee that tokens are immutable values, operations on tokens typically allocate a new token to represent the returned value.

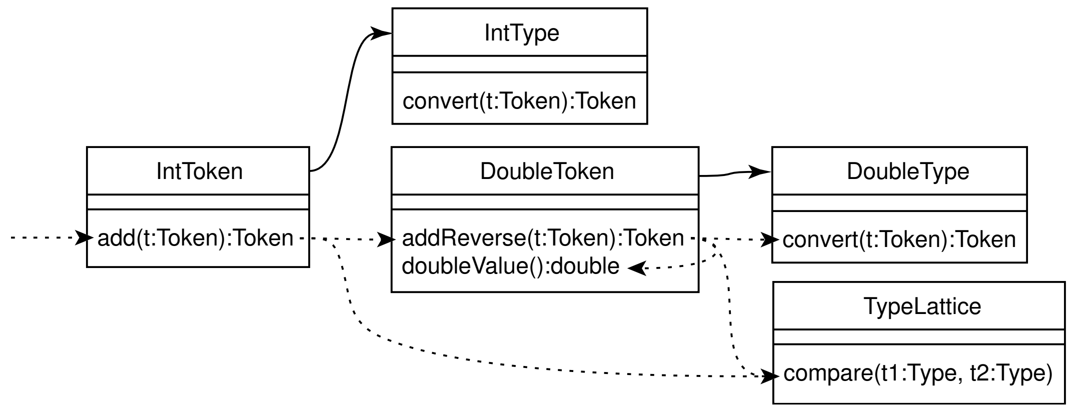


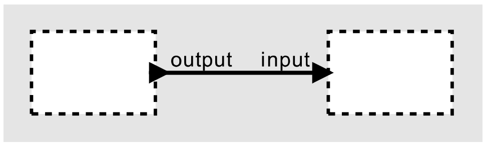
Figure 6.4: Typical indirection diagram for the data package. This diagram shows the indirection in an invocation of the `add()` method on an IntToken with a DoubleToken argument.

6.1.3 Type Checking

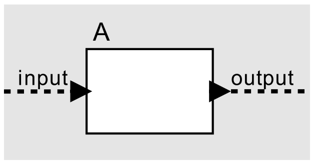
In addition to token operations, type conversions are also performed automatically during communication from one port to another. Ptolemy II includes a static type inference and checking system [78, 88, 103] that infers the types of ports and parameters to determine when type conversion should occur and statically ensures type safety in the presence of automatic type conversions.

This type system is based on type constraints expressed as inequality constraints on the lattice in Figure 6.3. These type constraints are implied by connections between ports, and by the operations implemented by actors. Type constraints in dataflow models typically take the form shown in Figure 6.5, where the type constraints for connections represent the presence of automatic type conversion and the type constraint for an actor gives the type of output ports in terms of the types of input ports and parameters.

Type checking is performed by collecting type constraints from the model and solving them for the least solution. A type error is reported if no solution to the type constraints exists, or ports are parameters are assigned the artificial type unknown. For type constraints of the form shown in Figure 6.5, a unique least solution to the constraints can be found efficiently as long as F_A is a monotonic function [87]. Figure 6.6 shows the application of type inference to a simple model.



Type(output) ≤ Type(input)



F_A(Type(input)) ≤ Type(output)

Figure 6.5: Type constraints in dataflow models. In general, F_A gives the types of output ports in terms of the types of input ports and parameters.

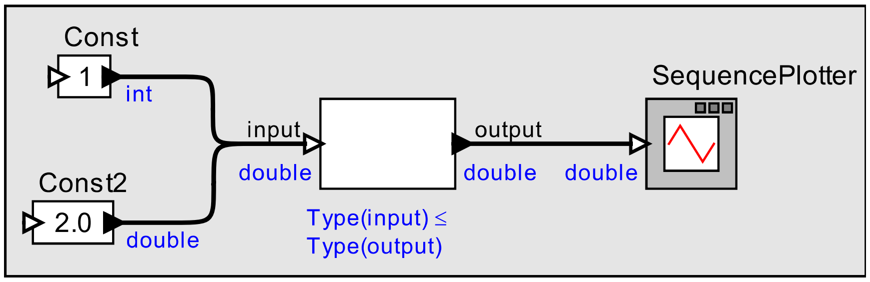


Figure 6.6: Type inference in a dataflow model. In this model, the value produced by Const would be automatically converted from and integer to a double before being received by input.

6.1.4 Parameters and Expressions

Configuration parameters in Ptolemy II models are represented by instances of the Parameter class. This class provides mechanisms for evaluation of parameter expressions and for caching and reevaluating the expression when necessary using a lazy evaluation strategy [43]. When a parameter is reconfigured, other dependent parameters are notified and a flag is set to indicate that the value must be recomputed. When the value is recomputed, the `attributeChanged()` method of the parameter's container is called and the flag is cleared. The `attributeChanged()` method is often used in actor classes to check for consistency between the values of different parameters.

Evaluation of expressions is performed by parsing the expression into an AST and traversing the AST to evaluate a token value for the root node. The Java code to perform the evaluation is architected using a Visitor design pattern [32]. Evaluating each node in the parse tree results in two virtual method invocations, in addition to other method calls to

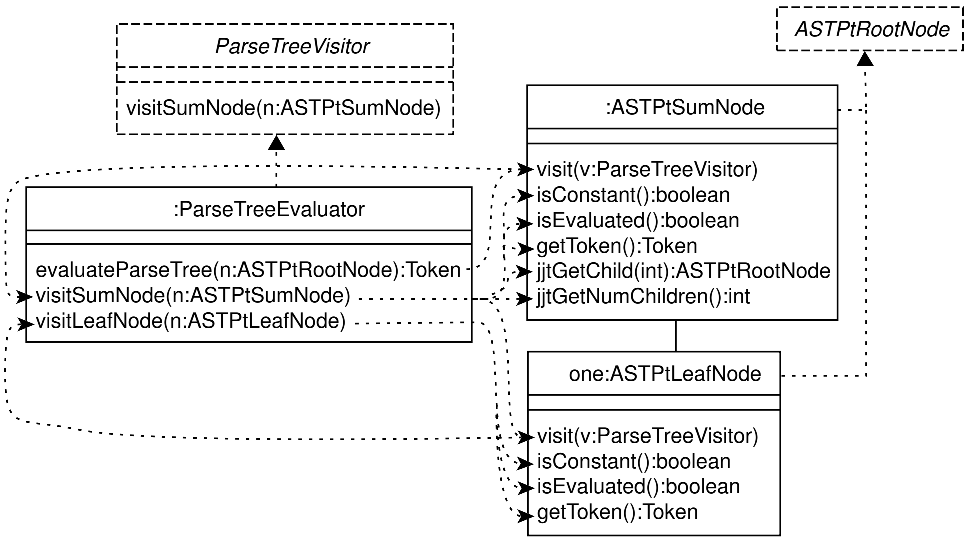


Figure 6.7: Indirection in parse tree evaluation for evaluating a expression containing a sum.

traverse the AST. An indirection diagram for this evaluation is shown in Figure 6.7.

6.1.5 Ports and Communication

The communication interface of an actor is represented by instances of the TypedIOPort class. The connections between these ports in a model are represented by instances of the TypedIORelation class. Each connection between an output port and an input port forms a communication channel, represented by an instance of a domain-specific class implementing the Receiver interface. Since modifications to the structure of a model are unusual, the receivers reachable from an port are cached to make data transport more efficient. As a result, sending or receiving data does not deal with relations, but only with the communication channels.

However, there is still a significant amount of processing that must be performed to transmit a single data value from one actor to another, as illustrated in Figure 6.8. Read access to the workspace is obtained, in order to ensure that run-time modifications to the model are not made. Run-time type checking in the `_checkType()` method ensures that actors respect type declarations. If necessary, run-time type conversion is performed by invoking the `convert()` method of the destination port. Lastly, the converted token is

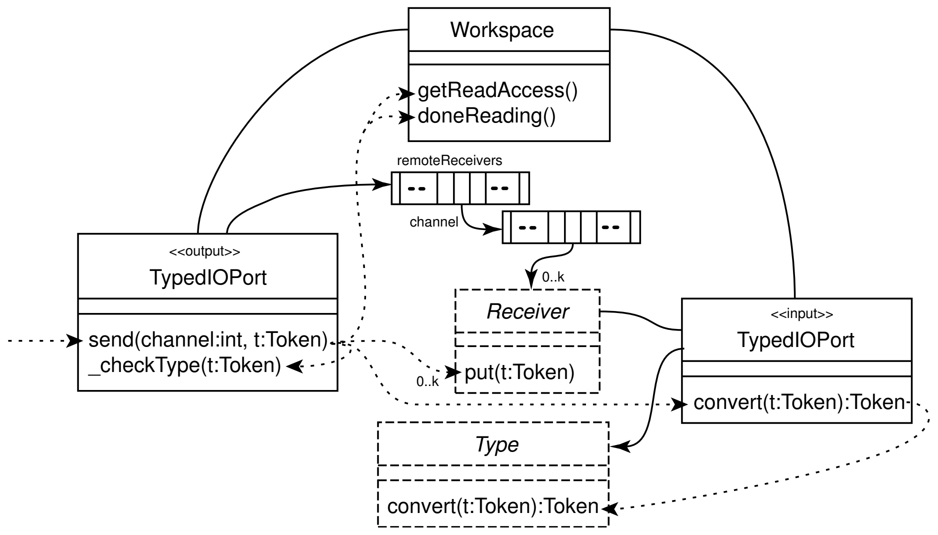


Figure 6.8: Indirection in sending data.

placed in the appropriate receiver via the `put()` method. The end result is a minimum of four class field access, two indexes into an array, and six method calls, in addition to domain-specific and type-specific code of unknown complexity. Similar overhead is incurred to retrieve the token using the `get()` method of the input port.

In Figure 6.8, notice that each indirection is present to support particular features in the Ptolemy II system. The Receiver interface is present to support domain-polymorphic actors, while the `convert()` and `_checkType()` methods support automatic type conversions. The arrays of receivers support addressing individual channels on each port through the first argument of the `get()` and `send()` methods and the broadcast of data to multiple input ports. Workspace synchronization enables modifications to models in a multi-threaded environment.

It is important to recognize that almost no model uses all of these features on every connection. In most models, output ports are connected to a single input port of the same type. Most models are not structurally reconfigured while executing. However, in some models these features allow for simpler models and more convenient specification. Unfortunately, to support these features in any model, all models must incur the same indirection overhead.

```
public class Const extends Source {
    public Const(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException {
        // The super class declares the output port.
        super(container, name);
        // Declare the value parameter
        value = new Parameter(this, "value");
        // and its default value.
        value.setExpression("1");
        // Set the type constraint of the output port.
        output.setTypeAtLeast(value);
        // Declare the graphical representation.
        _attachText("_iconDescription", "<svg>\n␣<rect␣x=\"0\"␣y=\"0\"␣"
                + "width=\"60\"␣height=\"20\"␣style=\"fill:white\"/>\n"
                + "</svg>\n");
    }
    public Parameter value;
    public void fire() throws IllegalActionException {
        super.fire();
        output.send(0, value.getToken());
    }
}
```

Figure 6.9: Complete Const actor class.

6.1.6 Actor Specifications

Specifications of actor behavior in Ptolemy II are given by subclasses of the TypedAtomicActor and TypedCompositeActor base classes. These subclasses, called *actor classes*, use the programming interfaces (APIs) provided by Ptolemy II to model actor behavior. These APIs provide an actor-oriented abstraction as long as actor classes respect a stylized form of Java.

A typical actor specification in shown in Figure 6.9. This class derives from the Source base class, which creates a port named output and provides default implementations of all actor methods. The constructor consists primarily of method invocations that create objects to represent the fixed interface of an actor. Although these objects are created only when an object is constructed, it is useful to think of the constructor code as declarations of actor structure. The call to the `_attachText()` method declares the representation of the actor in a visual editor. The `fire()` method implements the interesting behavior of this actor, which sends the value of the value parameter to the output port.

In general, actor classes can be written without any assumptions on the structure of a model by using the appropriate Ptolemy II APIs to dynamically discover ports and connections. Most actors, however, are written the style of the Const actor, where the ports and parameters of the actor are declared by the actor when the actor is created. As a result, only a relatively small number of methods defined by the Ptolemy II API need to be considered in actor classes. For example, Figure 6.10 shows the methods that are particular to the TypedIOPort class. However, only a small number are commonly used in actor classes.

When used correctly, the Ptolemy II APIs provide a robust actor-oriented semantics. One basic restriction is that actor classes interact with other actor classes only through the Ptolemy II APIs. Actor classes are expected to not declare `static` fields or methods for communicating with other actors. Additionally, although the Ptolemy II API provides mechanisms for retrieving actors in a model by name and manually traversing relations to other actors, actor classes are expected to be self-contained and not invoke these methods. Instead, actor classes declare public fields for keeping track of ports and parameters that form the actor's interface.

6.1.7 Model Specifications

In most cases, Ptolemy II models are not specified by manually writing Java code, but indirectly by constructing a model in a graphical editor or by writing an XML-style specification of a model using the Modeling Markup Language (MoML) [68]. Processing a MoML file results in the instantiation of the correct actor classes for each actor in the model and the invocation of other methods to create hierarchy and connections in the model. In effect, actor classes implement reusable software components which can be instantiated and composed in many different models.

6.2 Copernicus

Copernicus is a tool that generates Java code from a Ptolemy II model. The generated code has the same functional behavior as the original model, with the possibility of greatly improved performance. The construction of Copernicus was motivated primarily by the

boolean isDeeplyConnected(ComponentPort)
boolean isInput()
boolean isInsideLinked(Relation)
boolean isLinked(Relation)
boolean isMultiport()
boolean isOpaque()
boolean isOutput()
List insidePortList()
List insideRelationList()
List insideSinkPortList()
List insideSourcePortList()
int getWidth()
int getWidthInside()
int numInsideLinks()
int numLinks()
int numberOfSinks()
int numberOfSources()

Enumeration connectedPorts()
Enumeration deepConnectedInPorts()
Enumeration deepConnectedOutPorts()
Enumeration deepConnectedPorts()
Enumeration deepInsidePorts()
Enumeration insidePorts()
Enumeration insideRelations()
Enumeration linkedRelations()
List connectedPortList()
List deepConnectedInPortList()
List deepConnectedOutPortList()
List deepConnectedPortList()
List deepInsidePortList()
List linkedRelationList()
List sinkPortList()
List sourcePortList()
NamedObj getContainer()

(a) Query Structure

int moveToFirst()
int moveToLast()
int moveUp()
int moveDown()
int moveToIndex(**int** index)
void insertInsideLink(**int**, Relation)
void insertLink(**int**, Relation)
void liberalLink(ComponentRelation)
void link(Relation relation)

void setContainer(Entity)
void setInput(**boolean**)
void setMultiport(**boolean**)
void setName(String)
void setOutput(**boolean**)
void unlink(Relation)
void unlink(**int**)
void unlinkAll()
void unlinkAllInside()
void unlinkInside(Relation)
void unlinkInside(**int**)

(b) Modify Structure

InequalityTerm getTypeTerm()
List typeConstraintList()
Type getType()
boolean isTypeAcceptable()
void addTypeListener(TypeListener)
void removeTypeListener(TypeListener)

void setTypeAtLeast(InequalityTerm)
void setTypeAtLeast(Typeable)
void setTypeAtMost(Type)
void setTypeEquals(Type)
void setTypeSameAs(Typeable)

(c) Type System

Figure 6.10: Methods that can be invoked on the TypedIOPort class.

void broadcastClear()
Token get(**int**)
Token getInside(**int**)
Token[] get(**int**, **int**)
boolean hasRoom(**int**)
boolean hasRoomInside(**int**)
boolean hasToken(**int**)
boolean hasToken(**int**, **int**)
boolean hasTokenInside(**int**)

void broadcast(Token)
void broadcast(Token[], **int**)
boolean isKnown()
boolean isKnown(**int**)
boolean isKnownInside(**int**)
void send(**int**, Token)
void send(**int**, Token[], **int**)
void sendClear(**int**)
void sendClearInside(**int**)
void sendInside(**int**, Token)

(d) Actor Communication

void attributeChanged(Attribute)
Token convert(Token)
double getCurrentTime(**int**)
String toString()
boolean transferInputs()
boolean transferOutputs()

List sourcePortList(Receiver)
Receiver[][] deepGetReceivers()
Receiver[][] getInsideReceivers()
Receiver[][] getReceivers()
Receiver[][] getReceivers(IORelation)
Receiver[][] getReceivers(IORelation, **int**)
Receiver[][] getRemoteReceivers()
Receiver[][] getRemoteReceivers(IORelation)

(e) Miscellaneous

Figure 6.10: Methods that can be invoked on the TypedIOPort class.

desire to execute models of embedded software constructed using Ptolemy II in embedded Java environments. The Ptolemy II libraries alone consume several Megabytes of `.class` files, and require significant run-time memory allocation. Large memory allocation adds load to the garbage collector, reducing overall execution speed. Indirection adds overhead to each operations, further reducing execution speed. As a result, using Ptolemy II models directly in an embedded system in resource-constrained embedded systems.

Copernicus leverages the fact that the memory and processing requirements of Ptolemy II models are in most cases dominated by the organizational complexity assocated with generic and reusable actors. This complexity is an intrinsic part of Ptolemy II actors, even if the generic aspects of a component are not exercised through reconfiguration. Even in models that do perform reconfiguration, it is typically limited to a relatively small portion of most models. Copernicus specializes actors [81] using a combination of partial evaluation and generative programming to transform a model with Ptolemy II abstractions into self-contained Java code. Actor specialization allows access to design benefits of generic mechanisms, while incurring performance overhead only with these mechanisms are actually used in a model.

6.2.1 Code Generation from a Model

The code generator begins by generating a model suitable for specialization. Each actor class instantiated in the model is duplicated in order to specialize each actor independently. A new Java class is automatically generated for each hierarchical model. Constructor methods are automatically generated for each class that instantiate objects to represent ports and parameters contained by each actor and model. The resulting Java code can be executed using the Ptolemy libraries in much the same fashion as a model dynamically instantiated from a MoML description.

The bulk of the complexity in Copernicus arises from specialization transformations applied to the generated code. These transformations generate code that replaces method invocations of methods in the Ptolemy II APIs with specialized implementations. Methods on ports, parameters, and actors that are used to query the structure of the model are replaced with direct references to the correct object instances. Methods on ports that communicate data are replaced with domain-specific code for communication. Token objects are replaced with primitive Java code that is behaviorally identical but does not require object allocation. The resulting specialized code is entirely self-contained and does not depend on the Ptolemy II libraries.

6.2.2 Transformation Rules

We present the transformations applied in Copernicus in the style of rewrite rules. The left side of the rule is repeatedly matched against the code, and each match is replaced with the right side of the rule. In actuality, the transformations are implemented as transformations on abstract syntax trees, and we can generally interpret the rules below as tree rewrite rules. For conciseness, these rules have been written with a mixture of *syntactic code* representing Java syntax trees (in `typewriter` font) and *rewrite code* representing operations applied during code generation (in **boldface**). For convenience, we will assume that rewrite code for an object reference, such as **o**, corresponds to a reference to a uniquely identifiable Java object that can be statically determined. Put another way, `o.toString()` is the Java syntax tree corresponding to a method invocation on an unknown object, whereas **o.toString()** is the string resulting from invoking the `toString()` method on a particular object **o**. For Java primitive types, the automatically generated code should be obvious (numerical constants, String constants). Transformations have access to the functions listed below that generate appropriate syntax trees.

The following functions are available as compile-time operations:

- **objectReference**:NamedObj → AST (Returns an Abstract Syntax Tree that retrieves a runtime reference to the given input object.)
- **stringConstant**:String → AST (Returns an Abstract Syntax Tree for the give input string)
- **typeConstant**:Type → AST (Returns an Abstract Syntax Tree that creates a Type object that is equivalent to the given input type.

6.3 Actor Specialization

As mentioned previously, Java specifications of Ptolemy II actors define actor behavior in a generic way. In order to generate an efficient implementation from a specification, it is transformed into a new actor specification that is specialized to a particular *context*. Such a context includes, for instance, assignments of values to parameters and assignments of types to ports and parameters. While such a context could be specified explicitly, it is usually more convenient to use the concentrate on using the implicit context that actors acquire when composed in a model. In particular, the implicit context of an actor includes information about data types and constant parameters in a model that can be automatically inferred from a model.

This report considers four types of actor specialization: structural specialization, parameter specialization, type specialization, and domain specialization. The following sections describe, for each type of specialization, the possibilities for determining whether or not the appropriate context of an actor can change. In each case, an actor specification can

```
a = o.getContainer(s);  ::=  a = objectReference(o.getContainer(s));
a = o.getAttribute(s);  ::=  a = objectReference(o.getAttribute(s));
e = e.getEntity(s);  ::=  e = objectReference(e.getEntity(s));
p = e.getPort(s);  ::=  p = objectReference(e.getPort(s));
r = e.getRelation(s);  ::=  r = objectReference(e.getRelation(s));
```

Figure 6.11: Transformations applied during structural specialization.

be specialized if the appropriate context does not change. Additionally, parameter specialization can be performed even if parameter values are dynamically reconfigured by a modal model. In all cases, it is assumed that the model is not dynamically reconfigured by any means external to the model.

Note that we do not consider specialization of data generality from actors. In most cases, actors used in models of embedded systems operate on unknown data, since they are constantly receiving unknown data from sensors in the physical world. Hence, data generality seems crucial to the notion of an embedded system. However, in some cases it is useful to have actors internal to a model that produce sequences of constant or deterministic data. In such cases it seems possible that classical compiler optimizations, such as constant propagation and constant expression elimination [79] could be applied at the model level. Although they are not described here, these specializations seem straightforward to apply.

6.3.1 Structural Specialization

Structural specialization of a model replaces methods that are normally used to traverse hierarchy in a Ptolemy II model. The Ptolemy II base classes implement many of these methods using dynamic data structures, such as lists. Since the structure of a model is assumed to not change, then these data structures can be replaced with field references for each contained or containing object.

Basic structural specialization transformations dealing with methods for traversing the model hierarchy are shown in Figure 6.11. Structural specialization also replaces the methods shown in Figure (a) that query the structure between ports. Transformations for the most common methods are shown in Figure 6.12.

`i = p.getWidth();` ::= `i =` **p.getWidth()**

`i = p.numberOfSinks();` ::= `i =` **p.numberOfSinks();**

`i = p.numberOfSources();` ::= `i =` **p.numberOfSources();**

`b = p.isInput();` ::= `b =` **p.isInput();**

`b = p.isOutput();` ::= `b =` **p.isOutput();**

`b = p.isMultiport();` ::= `b =` **p.isMultiport();**

Figure 6.12: Domain-independent transformations for specializing connections.

```
Port input1, input2, input3, output;
public void fire() {
    for(i = 0; i < input1.getWidth(); i++) {
        if(input1.hasToken(i)) {
            output.send(0, input1.get(i));
        }
    }
    for(i = 0; i < input2.getWidth(); i++) {
        if(input2.hasToken(i)) {
            output.send(0, input2.get(i));
        }
    }
    for(i = 0; i < input3.getWidth(); i++) {
        if(input3.hasToken(i)) {
            output.send(0, input3.get(i));
        }
    }
}
```

Figure 6.13: A merge actor.

The transformations above can result in significant code simplification for some actor specifications. As an example, Figure 6.13 shows an actor that transmits data received from any input port to its output port. This actor is specified in such a way that each of the three input ports may or may not be connected. If any of the inputs are not connected, then the corresponding loop will never actually consume any data. The above transformations enable the loop conditions to be statically evaluated, allowing for loop unrolling and elimination of any loops corresponding to ports with zero width.

6.3.2 Parameter Specialization

Parameter specialization transforms an actor specification with unspecified parameter expressions into a specification where parameter expressions are fixed. Two different specializations are possible, depending on whether parameters are constant or not. If a parameter value is constant, then its value is fixed throughout execution of a model and queries for the value of constant parameters can be replaced with the constant value of the parameter. If a parameter is not constant then code can be generated from the expression that allows the parameter value to be evaluated without the overhead of traversing a parse tree at execution time.

In the context of this chapter, the reconfiguration analysis in Chapter 4 determines the binding times of parameters in the sense of a partial evaluator. Constant parameters are similar to variables with static binding time, while parameters that are not constant are similar to variables with dynamic binding time. In addition to identification of inherently constant parameters, reconfiguration analysis also provides information about when parameters are reconfigured. This information can be used to further specialize the evaluation of not-constant parameters.

Replacing Parameters

Objects representing parameters can be specialized using the transformation rules in Figure 6.14. These rules implement a *lazy* evaluation strategy [43], which is essentially identical to the algorithm implemented by the Parameter class. The generated code, however, is specialized to a particular parameter expression and dependencies between parameters. Prior to applying these transformations, fields are added to the actor class to represent the value of the parameter and whether or not the value is valid. These fields are returned by the **tokenField** and **isValidField** methods, which are functions that return an AST for querying the correct field for any parameters.

The lazy evaluation strategy, however, has several drawbacks. Every parameter requires an additional field that represents the validity of the value which must be checked whenever the parameter is queried. For frequently changing parameters, the validity check will usually returns `TRUE`, while for infrequently changing parameters, this check usually return

```
s = p.getName();  ::=  s = stringConstant(p.getName());

s = p.getFullName();  ::=  s = stringConstant(p.getFullName());

s = p.getType();  ::=  s = typeConstant(p.getType());

s = p.getExpression();  ::=  p.getToken().toString();

p.setExpression(s);  ::=  error

t = p.getToken();  ::=  if(!objectReference(p.getContainer()).isValidField(p)) {
                              objectReference(p.getContainer()).evaluateMethod(p)
                              attributeChanged(p)
                              objectReference(p.getContainer()).isValidField(p) = TRUE;
                          }
                          t = objectReference(p.getContainer()).tokenField(p);

p.setToken(t);  ::=   objectReference(p.getContainer()).tokenField(p) =
                          (tokenField(p).type)typeConstant(p.getType()).convert(t);
                      foreach x in sort(dependents(p)) {
                          objectReference(x.getContainer()).isValidField(x) = false;
                      }
                      attributeChanged(p);
```

Figure 6.14: Transformations applied during dynamic parameter specialization.

`FALSE`. Although many processor architectures include branch prediction hardware that can adapt to the frequency branches are taken, it is still preferable to produce code where such prediction is not necessary.

Fortunately, reconfiguration analysis provides exactly the information necessary to specialize actors more effectively. Parameters which are constant or constant over firings of an actor can be replaced with simpler code than the generic lazy evaluation strategy. These transformations are shown in the following sections.

Replacing Constant Parameters

Objects representing constant parameters can be more effectively specialized by computing the constant value at compile time and replacing accesses to the parameter with the constant value. Primarily, this results in the replacement of invocations of the parameter's `getToken()` method, with a reference to a token object. Since tokens are immutable

```
t = p.getToken();  ::=  t = objectReference(p.getContainer()).tokenField(p);

p.setToken(t);  ::=   objectReference(p.getContainer()).tokenField(p) =
                          (tokenField(p).type)typeConstant(p.getType()).convert(t);
                      attributeChanged(p);
```

Figure 6.15: Transformations applied during parameter specialization.

```
public TypedIOPort input, output;
public Parameter arrayLength;
public void fire() {
    int length = ((IntToken)arrayLength.getToken()).intValue();
    Token[] valueArray = input.get(0, length);
    output.send(0, new ArrayToken(valueArray));
}
```

Figure 6.16: Original code from SequenceToArray.

objects, the expense of runtime allocation is reduced by creating the tokens during initialization and storing a reference to the token in an automatically created field. The field is initialized through an invocation of the `setToken()` method when the model is initialized. The transformations are shown in Figure 6.15. Note that no field is necessary to record the validity of the parameter value.

As an example, consider the SequenceToArray actor specification shown in Figure 6.16. This actor consumes a number of tokens determined by the value of the arrayLength parameter and aggregates them into a single array token. This specification, specialized to a constant arrayLength parameter value of 8, is shown in Figure 6.17.

Since tokens represent immutable values, some method invocations, such as the `intValue()` method call in Figure 6.17, can also be replaced using constant propagation. The result is shown in Figure 6.18. In this case, since the parameter is not used elsewhere in the actor specification, the field and token creation are dead and can also be removed (see Section 6.5.3).

```
public TypedIOPort input, output;
public IntToken arrayLength_value = new IntToken(8);
public void fire() {
      int length = arrayLength_value.intValue();
      Token[] valueArray = input.get(0, length);
      output.send(0, new ArrayToken(valueArray));
}
```

Figure 6.17: The SequenceToArray actor after specialization with arrayLength = 8.

```
Port input, output;
public void fire() {
      int length = 8;
      Token[] valueArray = input.get(0, length);
      output.send(0, new ArrayToken(valueArray));
}
```

Figure 6.18: The SequenceToArray actor after additional specialization with arrayLength = 8.

```
t = p.getToken();  ::=  t = objectReference(p.getContainer()).tokenField(p);

p.setToken(t);  ::=   objectReference(p.getContainer()).tokenField(p) =
                          (tokenField(p).type)typeConstant(p.getType()).convert(t);
                      attributeChanged(p);


After reconfiguration at quiescent points of actor A  ::=
   P = {p : ⌊p⌋ = a}
   foreach p in sort(P) {
       objectReference(p.getContainer()).evaluateMethod(p)
   }
```

Figure 6.19: Transformations for parameter specialization using the Least Change Context evaluation strategy.

Replacing Other Parameters

Parameters which are constant over the firings of an actor are guaranteed to not be reconfigured during those firings. As a result, it is sufficient to guarantee the validity of a parameter that is constant over firings of an actor a only at the quiescent points of actor a. This evaluation strategy will be called a *least change context* evaluation strategy. The corresponding transformations are shown in Figure 6.19.

Although the least-change context evaluation strategy alone is not optimal, since it may re-evaluate parameter values even when no reconfiguration has been performed, it does not require separate checks for validity. A *hybrid* lazy/least-change context strategy is also possible, where validity checks are inserted after reconfiguration occurs. Due to reconfiguration analysis, fewer validity checks are usually required in such a strategy than in the purely lazy evaluation strategy, although code to set validity flags during reconfiguration is still necessary. Depending on how often parameter values are reconfigured and the complexity of computing new parameter values, either the least-change context evaluation or hybrid strategies may be preferable.

```
t = p.getToken();  ::=  t = objectReference(p.getContainer()).tokenField(p);
p.setToken(t);  ::=    objectReference(p.getContainer()).tokenField(p) =
                           (tokenField(p).type)typeConstant(p.getType()).convert(t);
                       attributeChanged(p);
```

Figure 6.20: Transformations applied during dynamic parameter specialization.

6.3.3 Type Specialization

Actor specifications in Ptolemy II are often type polymorphic, allowing them to operate equally well on integers, doubles, or more complex structured data types. This polymorphism is abstracted by the ptolemy.data package, which represents a set of common type-independent operations. Although actors are generically typed, in most models actors only ever receive or produce data of a single exact token type. By applying type-inference applied to actor classes, exact types can usually be inferred where they are not completely specified. *Type specialization* transforms a type-polymorphic specification to a specification that only operates on a single type. When combined with token unboxing, type specialization removes the indirection caused by using the ptolemy.data package.

Inferring Token Types in Java Actor Specifications

In a model, the Ptolemy II type system infers the types of ports using type constraints for actors that are specified by actor writers and assumed to be correct. Types assigned to ports and parameters determine where in a model automatic type conversions occur. Propagating these types through Java code requires a different type inference procedure for several reasons. Most importantly, type conversions cannot always be inserted into an actor class while preserving the behavior of the actor code. Additionally, although tokens represent immutable values, some data structures used in actor code, such as arrays of tokens, are mutable. Inference of the types of such structures must consider the possibility of aliased references. Lastly, Java allows for local variables and fields to reference different token types in different control paths. To recover exact types, such cases must be eliminated.

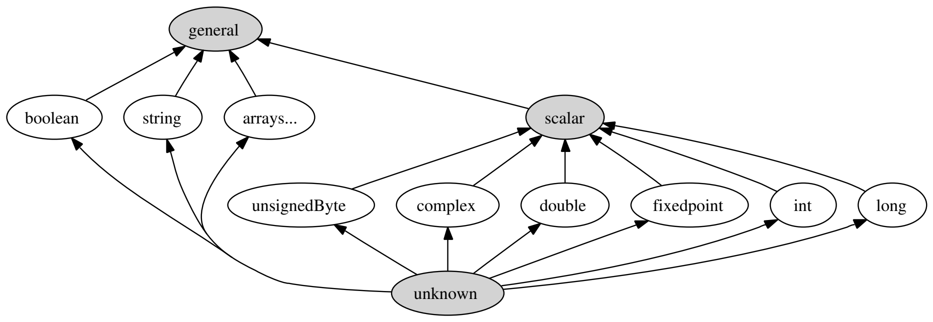


Figure 6.21: The type lattice used for inferring token types in Java code.

Basic Type Inference

Since automatic type conversions cannot be inserted into Java code, the type lattice in Figure 6.3 does not represent the correct relationship between types. Type inference for Java code is performed using the type lattice shown in Figure 6.21. This lattice is similar to the automatic conversion type lattice, except that exact types are considered to be incomparable.

The primary token type inference algorithm computes a type for each local variable and class field that refers to a token object, including arrays of tokens. The types of ports and parameters of the actor are fixed based on types inferred from the model, as are the types of newly created tokens. Assignments require that types on both sides of the assignment are equal, in order to capture constraints between aliased objects. The type inference rules are shown in Figure 6.22.

Types and Control Flow

It is important to note that, unlike the type checking systems applied to Java source code or Java bytecode, this system requires variables to have the same type at all points in a method. If a variable references different token types at different points in a method, then no solution will be found to the constraints above. In contrast, the type system applied to Java source code during compilation allows variables to refer to objects of different classes, as long as they are all subclasses of the declared type. The type system applied to Java bytecode is also more lenient, since it allows a variable to be assigned different exact

$$\texttt{A = B;} \implies \mathit{Type}(A) = \mathit{Type}(B)$$
$$\texttt{A = (foo)B;} \implies \mathit{Type}(A) = \mathit{Type}(B)$$
$$\texttt{A[i] = B;} \implies \mathit{Type}(A) = \mathit{Type}(B)$$
$$\texttt{A = B[i];} \implies \mathit{Type}(A) = \mathit{Type}(B)$$
$$\texttt{A[i] = B[i];} \implies \mathit{Type}(A) = \mathit{Type}(B)$$
$$\texttt{A = P.getToken();} \implies \mathit{Type}(A) = \mathit{Type}(P)$$
$$\texttt{A = P.get(i);} \implies \mathit{Type}(A) = \mathit{Type}(P)$$
$$\texttt{T = P.getType();} \implies T = \mathit{Type}(P)$$
$$\texttt{A = T.convert(B);} \implies \mathit{Type}(A) = T$$

Figure 6.22: Rules for inferring token type constraints in Java code.

types at different points in a program. The type system used here emphasizes the discovery of exact types in the presence of type declarations and considers code that assigns multiple token types to a single variable to be illegal.

As an example, consider the specification of the Ramp actor in Figure 6.23, where the value of the `init` parameter is an IntToken, and the `step` parameter contains a DoubleToken. Based on the type constraints declared by this actor, the type of the output port is double. On the first firing, the field `_state` refers to an IntToken, which is the value of the `init` parameter. Since the output port has type double, the IntToken is converted to a DoubleToken in the process of being sent. After the `postfire()` method is invoked, the `_state` field refers to a DoubleToken, which results from adding the initial IntToken to the value of the `step` parameter (a DoubleToken). The token type inference system will flag this as a type error, although it is valid Java code and executes without error in simulation.

There are different ways of modifying the Ramp actor class to obtain code with exact token types. One solution is to strengthen the type constraints on the actor, declaring that the types of the parameters and the types of the output port must all be the same. In this case, the value of the init parameter will be automatically converted into a DoubleToken before being queried, and exact types are present. Another solution, shown in Figure 6.24, is to manually insert code into the `initialize()` method to convert the value of the initial token to the type of the output port. Note that neither of these solutions can be applied automatically, since they have the potential to change the behavior of the program.

```
TypedIOPort output;
Parameter init;
Parameter step;
private Token _state;
public void initialize() {
    _state = init.getToken();
}
public void fire() {
    output.send(0, _state);
}
public void postfire() {
    _state = _state.add(step.getToken());
}
```

(a)

$$Type(output) = double$$
$$Type(init) = int$$
$$Type(step) = double$$
$$Type(\_state) = Type(init)$$
$$Type(output) = Type(\_state)$$
$$Type(\_state) = F_{add}(Type(\_state), Type(step))$$

(b)

Figure 6.23: A Ramp actor specification (a) which does not have exact token types, as shown by the inconsistent type constraints in (b).

```
TypedIOPort output; // double
Parameter init; // int
Parameter step; // double
private Token _state; // double
public void initialize() {
    _state = step.getType().convert(
                    init.getToken());
}
public void fire() {
    output.send(0, _state);
}
public void postfire() {
    _state = _state.add(step.getToken());
}
```

(a)

$$Type(output) = double$$
$$Type(init) = int$$
$$Type(step) = double$$
$$Type(\_state) = Type(step)$$
$$Type(output) = Type(\_state)$$
$$Type(\_state) = F_{add}(Type(\_state), Type(step))$$

(b)

Figure 6.24: A Ramp actor specification (a) with exact token types, as shown by the consistent type constraints in (b).

```
public TypedIOPort input1, input2, output;
public void fire() {
    super.fire();
    if ((input1.getType() instanceof ArrayType) &&
        (input2.getType() instanceof ArrayType)) {
        _arrayFire();
    } else if ((input1.getType() instanceof UnsizedMatrixType) &&
            (input2.getType() instanceof UnsizedMatrixType)) {
        _matrixFire();
    } else {
        throw new IllegalActionException("Invalid␣types");
    }
}
```

Figure 6.25: Code from the DotProduct actor.

Type-controlled Conditionals and Recursion

In Figure 6.23, the code cannot be specialized since no solution to the type equations can be found. However, the lack of a solution does not necessarily indicate that specialization is impossible. For instance, in actors where code is only conditionally executed, no solution may exist due to type constraints from code that is never executed. This section presents a more detailed type analysis that can be used to identify dead code, with the goal of obtaining code which can be type specialized. The more detailed analysis differs primarily in that it computes the type of variables at every statement.

For example, consider the code in Figure 6.25 taken from the DotProduct actor. This code performs two completely different operations, depending on the types of the input ports. The type of data received is determined at run time, using Java's `instanceof` operator. However, the type of the input ports can be statically determined through type inference on the model. By propagating this information through the actor class, the `instanceof` operations can be evaluated as constant expressions and the unexecuted branch eliminated.

A more complex example is the actor specification shown in Figure 6.26. This actor takes an input, which can be either an ArrayToken or numeric token, and multiplies it by the parameter `factor`. If the input is an ArrayToken, then the output is also an ArrayToken. In this case, the `_scaleOnLeft()` is called recursively on each element of the array, if necessary.

```
TypedIOPort input,output; // array(double)
Parameter factor; // double
public void fire() {
    if (input.hasToken(0)) {
        Token in = input.get(0);
        Token factorToken = factor.getToken();
        Token result = _scaleOnLeft(in, factorToken);
        output.send(0, result);
    }
}

private Token _scaleOnLeft(Token input, Token factor) {
    if(input instanceof ArrayToken) {
        Token[] argArray = ((ArrayToken)input).arrayValue();
        Token[] result = new Token[argArray.length];
        for (int i = 0; i < argArray.length; i++) {
            result[i] = _scaleOnLeft(argArray[i], factor);
        }
        return new ArrayToken(result);
    } else {
        return factor.multiply(input);
    }
}
```

Figure 6.26: An actor that scales its input.

In object-oriented partial evaluation systems, this situation is often addressed by duplicating the method for each possible input type, a technique known as *polyvariant specialization*. However, this technique generally requires complex inter-procedural analysis that crosses method calls, in order to discover the required types. Rather than building such an analysis, Copernicus inlines non-recursive method invocations that have Token classes as arguments or return types. This approach is simpler, although slightly less robust since it assumes that recursion is governed according to the argument type (as in the Scale actor) and inlining code can potentially increase code size.

Type Specialization Transformations

Inference of token types within Java code leads to several automatic transformations. Primarily, Java fields which maintain the state of the actor, such as the field `_state` in Figure 6.24, can be given new Java types that more accurately reflect the data they reference.

Similar transformations can be performed on Java arrays of tokens. These transformations often require the insertion of Java casts to ensure that the specification is still properly typed under the Java type system. Although these transformations do not significantly modify the behavior of the code, they enable the unboxing of tokens, described in Section 6.4.3.

6.4 Domain Specialization

Domain specialization transforms an actor specification that is constructed using the domain-polymorphic interfaces into a new actor specification that is fixed to a particular model of computation. Unlike other forms of actor generality, domain polymorphism seems unique in that it does not enable useful dynamic reconfiguration of a model. That is, we have not seen instances where is it useful to change the model of computation of a model while the model is executing. Hence, we assume that the context of an actor specifies a single, fixed model of computation.

Domain specialization effectively replaces the director and receiver objects in a model. This results in more efficient execution since the code in the Director and Receiver classes can be specialized to the structure of a particular model. Additionally, the communication methods of the Port class can be replaced with domain specific code, eliminating an unnecessary level of indirection.

Copernicus implements domain specialization using specialized code generators for each model of computation. Given the complexity of code, particularly inside the Director class for a model of computation, specialized code generators provide a pragmatic path to more efficient code at the expense of duplicating some of the logic that already exists in the domain-specific classes. Additionally, the use of specialized code generators allows for implementation-specific communication libraries to be used, where possible.

6.4.1 Dataflow Scheduling is Model Specialization

Copernicus primarily supports domain specialization for sequentialized execution of *synchronous dataflow* (SDF) models. We concentrate on these models to get a sense of the performance limits of the specialization approach, since SDF models can be implemented efficiently. In a practical design flow, sequentialized execution of SDF models would likely be combined with concurrent execution and other timed and untimed models of computation. Copernicus is architected to allow exactly such a scenario.

Part of the efficiency of SDF models can be seen easily by considering SDF as a specialization of generic, dynamically scheduled dataflow models. Figure 6.27 shows a generic algorithmic framework for run-time scheduling of dataflow models. The scheduling logic is encapsulated in the selectNextActor function, which returns the next actor to execute. The code for executing a particular finite-length dataflow schedule can be derived from this algorithm by generalized partial evaluation, including loop unrolling and function inlining [46].

The key constraint on SDF models is that the consumption and production functions are no longer functions of state, enabling the elimination of the consumes function, which never changes. This allows the selectNextActor function to be partially evaluated with respect to the consumes function, resulting in a function of only the number of tokens in each channel. The loops can be unrolled, taking care to unroll the while loop only as necessary. At this point, all functions have only constant arguments and can be evaluated at compile time, resulting in the code in Figure 6.28. This specialization could be performed automatically by a generalized partial evaluator, given suitable formulation of the selectNextActor function.

```
Given: $\mathbf{A}, \mathbf{S}, \mathbf{C}$,
        consumption : $\mathbf{S} \to \mathbf{A} \to \mathbf{C} \to \mathbb{N}$,
        production : $\mathbf{S} \to \mathbf{A} \to \mathbf{C} \to \mathbb{N}$,
        initialState : $\mathbf{A} \to \mathbf{S}$,
        fire : $\mathbf{A} \to \mathbf{S} \to \mathbf{S}$,
        selectNextActor : $(\mathbf{C} \to \mathbb{N}) \to (\mathbf{A} \to \mathbf{C} \to \mathbb{N}) \to \mathbf{A} \to \mathbb{B}$

for $c \in \mathbf{C}$ do
    tokens($c$) = 0
od
for $a \in \mathbf{A}$ do
    state($a$) = initialState($a$)
od
for $a \in \mathbf{A}$ do
    consumes($a$) = consumption(state($a$), $a$)
od
($a$, $done$) = selectNextActor($tokens$, $consumes$)
while (!$done$) do
    for $c \in \mathbf{C}$ do
        tokens($c$) = tokens($c$) − consumption(state($a$), $a$) + production(state($a$), $a$)
    od
    state($a$) = fire($a$, state($a$))
    consumes($a$) = consumption(state($a$), $a$)
    ($a$, $done$) = selectNextActor($tokens$, $consumes$)
od
```

Figure 6.27: An algorithm for sequential execution of dataflow models.

```
Given: $\mathbf{A}, \mathbf{S}$, fire : $\mathbf{A} \to \mathbf{S} \to \mathbf{S}$

while (TRUE) do
    state(FIR) = fire(FIR, state(FIR))
    state(FIR2) = fire(FIR2, state(FIR2))
    state(FIR2) = fire(FIR2, state(FIR2))
    state(FIR2) = fire(FIR2, state(FIR2))
od
```

Figure 6.28: Specialized execution code for the model in Figure 2.5, derived from the code in Figure 6.27 by specialization.

6.4.2 Domain Specialization Transformations

In an SDF model, all execution and communication can be statically scheduled [11]. Appropriate code is generated directly from the Synchronous Dataflow schedule to execute the fire method of each actor and communication between ports is implemented using fixed length arrays and circular addressing. To reduce the buffering requirements, the communication buffers are shared in cases where data is broadcast to multiple receiving ports. Invocations of the `get()` and `send()` are replaced with array reads and writes and circular buffer addressing. The length of each array is statically computed by simulating the execution of the schedule. The corresponding transformations are shown in Figure 6.29. The **bufferArrayField** function returns an AST that references an array of communication buffers for a particular port, while the **indexArrayField** function returns an AST that references an array of the circular addressing indexes used to index into that buffer.

Note that if the channel argument to the `get()` or `send()` method can be statically determined, then indexing into the array of buffers for each port is not required. In such cases, Copernicus creates an additional field for each communication channel that points directly to the correct buffer, eliminating an array index operation. An additional optimization can be performed if the length of the buffer for a particular operation is known to be one. In such a case, the circular addressing operation is trivial and can be eliminated entirely.

The code in Figure 6.30 illustrates the resulting transformed code for the SequenceToArray actor in the example model. This code contains references to the arrays of tokens for the input and output buffers. It also contains a reference to the array of indices into the input buffer which is updated as data is read from the buffer. Note that the channel indexes are known statically, eliminating the need for an array of buffers, and that no array of indices is created for the output buffer, which contains only a single location.

6.4.3 Token Unboxing

The boxing and unboxing of data is a well-known technique used in the implementation of functional languages, such as ML [70]. In functional languages, the goal of unboxing is to be able to pass numeric types to type-polymorphic functions. The functions themselves are written to handle arbitrary objects, but are unable to handle numeric values. *Boxing*

```
b = p.hasToken();  ::=  b = true;

b = p.hasRoom();  ::=  b = true;

p.send(c,t);  ::=   a = objectReference(p.getContainer());
                    index = a.indexArrayField(p)[c];
                    buffer = a.bufferArrayField(p)[c];
                    buffer[index] = t;
                    a.indexArrayField(p)[c] = (index + 1) % buffer.length;

p.broadcast(t);  ::=   foreach c between 0 and p.getWidth() {
                           a = objectReference(p.getContainer());
                           index = a.indexArrayField(p)[c];
                           buffer = a.bufferArrayField(p)[c];
                           buffer[index] = t;
                           a.indexArrayField(p)[c] = (index + 1) %
                       buffer.length;
                       }

t = p.get(c);  ::=   a = objectReference(p.getContainer());
                     index = a.indexArrayField(p)[c];
                     buffer = a.bufferArrayField(p)[c];
                     t = buffer[index];
                     a.indexArrayField(p)[c] = (index + 1) % buffer.length;
```

Figure 6.29: Domain specialization transformations for actors in SDF models. The `sendInside()` and `getInside()` methods are replaced similarly to the `send()` and `get()` methods.

```
// The buffer for the input port.
DoubleToken[] _relation_0_double;
// The current read index for the input port.
int[] _index_input;
// The buffer for the output port.
ArrayToken[] _relation3_0__double_;
public void fire() {
    // Code replacing the input.get() method
    DoubleToken[] doubletokens = _relation_0_double;
    int index = _index_input[0];
    Token[] tokens = new Token[8];
    for (int i = 0; i < 8; i++) {
        tokens[i] = doubletokens[i];
        index = ++index % 8;
    }
    _index_input[0] = index;
    ArrayToken arraytoken = new ArrayToken(tokens);
    // Code replacing the output.send() method
    ArrayToken[] arraytokens = _relation3_0__double_;
    arraytokens[0] = arraytoken;
}
```

Figure 6.30: The SequenceToArray actor, with optimized communication.

refers to the process of automatically encapsulating a numeric value in a wrapper object so that it can be passed to such a type-polymorphic method. When the number is eventually passed to another method that requires the numeric value, it is automatically removed from its wrapper through *unboxing*. This transformation happens within the execution engine for the language and is totally transparent to the programmer.

Copernicus performs a transformation similar to unboxing: it replaces token objects (an abstract wrapper for a data object) with the value that the token contains. Similarly, operations on the token (i.e., method calls) are replaced with native numeric operations. For instance, the `IntToken.add()` method, which adds the values contained in two integer wrapper objects, is replaced with a simple integer addition. In most Java implementations, this greatly reduces the overhead involved in the operation. More importantly, the overhead of allocating and garbage collecting the wrapper object for the result is also eliminated.

It is important to notice that token unboxing is not possible in the presence of type-polymorphic actor specifications. Token unboxing is possible during code generation because the Ptolemy II type system emphasizes models where types are exactly determined and type-polymorphic actor specifications have been specialized to those exact types.

So, for a particular type of token, which native numerical type and operations should it replaced with? One possibility is to use a fixed and hardcoded replacement relation between a type of token and a native numerical type [95]. Unfortunately, this limits the ability to add new data types to the Ptolemy II framework, as the operations for each token must be essentially reimplemented in the code generation framework. We must also have some way of transforming structured token types that are not directly replaced with native types. This is not easily handled by a small set of hand-written rules.

We have implemented a technique for transforming tokens that does not rely on hand-written replacement rules. Instead of reimplementing each token operation, we make use of the specification of each token operation that already exists in the corresponding token class. Wherever a method is invoked on a token class, this method is inlined from the correct token class. Each token variable and field that refers to a token is replaced with variables and fields corresponding to the fields of the token class. Arrays of tokens are replaced with multiple arrays for each field of the token class. Additionally, a boolean field is created that tracks whether the original token reference is `null`. This flag is used to properly replace comparisons between the token and `null`. Note that because objects and arrays in Java have a 16 byte overhead for the object header, token unboxing applied to short arrays of tokens can result in the instantiation of more objects and a small increase in memory usage.

Token unboxing is similar to object inlining transformations [26, 27]. These transformations are based on analysis to detect *one-to-one contained* objects. The data representation for such objects can be optimized to eliminate an object indirection. As noted by Laud [60], object inlining can be easily applied when objects are *constant* and cannot be modified after creation, since aliasing and side effects can generally be ignored. Token unboxing can be safely applied to token classes, since these classes are constant to reflect the abstraction of immutable tokens. The relationship between immutable objects and aliasing is also leveraged by Guava [6] to optimize object allocation for *value objects* that are copied on assignment. If a value object is immutable, then copying need not be performed, since aliasing cannot cause side effects.

Token unboxing is generally effective for all numeric token types. Furthermore, it does not preclude optimized transformations for specific numeric types, such as those for fixed-point types [52], or for record types. It is also applicable for structured types as well, such as arrays and records. For instance, the ArrayToken class aggregates a set of other tokens and indexes them using integers. Since one field of the class contains an array of other tokens, unboxing the array token replaces it with an array of tokens. These tokens (regardless of their type) can then be unboxed by applying the above procedure recursively.

6.5 Application Extraction

Copernicus includes several Java-level static analyses for optimizing the resulting generated code. These analyses and the transformations that use them resemble those used in applications extraction [94]. The analyses and transformations presented here all rely on the fact that Java code generated by Copernicus is self contained and unknown libraries are not loaded dynamically. The first static analysis detects invocations of methods whose return value is not used and can be guaranteed to have no side effects. Such method invocations often arise from accessor methods whose return values are no longer needed after specialization. The second static analysis estimates the set of reachable methods which can be invoked during execution of the code. This analysis is used to automatically prune classes with no reachable methods and to remove unreachable methods from classes that cannot be pruned. Based on the reachable classes, Copernicus creates a self-contained `.jar` archive for the generated code.

6.5.1 Reachable Method Analysis

A Java method is *reachable* if can be invoked through a sequence of method calls from program's `main()` method. Specialization often results in many unreachable methods and removing such methods can greatly reduce the size of generated code. Reachable method analysis computes a function **isReachable** : *methods* $\rightarrow \mathbb{B}$. This function is defined by the least solution to the following set of constraints, computed on the lattice of boolean values $\mathbb{B}$ where `FALSE` $\leq$ `TRUE`:

- m_1 invokes $m_2 \implies \textbf{isReachable}(m_1) \leq \textbf{isReachable}(m_2)$
- m is a program entry point $\implies$ `TRUE` $\leq$ **isReachable(m)**

In practice, the implementation of reachable method analysis is slightly simplified to give good results in a reasonable amount of time. Primarily, method invocation through reflection can enable invocations of any method in a program, often resulting in every method in a program being reachable. Copernicus assumes that reflection only occurs within the standard Java libraries and that methods in generated code are not invoked through reflection (other than the standard `public void main()` entry point). Secondarily, since analyzing the standard Java libraries can be expensive, Copernicus assumes that every standard Java method is reachable. Reachability constraints from method invocations originating from the standard libraries are captured by assuming that methods that implement standard Java interfaces are program entry points.

Copernicus uses reachable method analysis to remove any methods **m** that are not reachable, i.e., **isReachable(m)** = `FALSE`. This transformation is safe because the generated Java code is known to be self-contained and Java's dynamic class loading and reflection facilities are not used in the generated code.

6.5.2 Side-Effecting Method Analysis

A Java method has *side effects* if it is capable of performing one of the following primitive operations:

- Store a value in a field of an object.
- Store a value in a static field of a class.
- Store a value into an array.
- Perform thread synchronization. (i.e., the method is declared `synchronized`, or contains a `synchronized` block)

Additionally, a method has side effects if it invokes any method that has side effects. Side effect analysis computes a function **hasSideEffects** : *methods* $\rightarrow \mathbb{B}$. This function is defined by the least solution to the following set of constraints, again computed on the lattice of boolean values $\mathbb{B}$ where $\mathtt{FALSE} \leq \mathtt{TRUE}$:

- m_1 can be invoked by $m_2 \implies \mathbf{hasSideEffects}(m_1) \leq \mathbf{hasSideEffects}(m_2)$
- m can perform a primitive side effecting operation $\implies \mathtt{TRUE} \leq \mathbf{hasSideEffects}(m)$

The practical implementation of side-effect analysis makes use of two simplifications. Primarily, only reachable methods need to be considered in the analysis, which limits the domain for which the **hasSideEffects**() function needs to be computed and reduces the set of methods that can be invoked at any invocation site. Secondarily, native methods and methods in the standard Java libraries are assumed to always have side effects. This restriction avoids analyzing the code of such methods to determine what methods can be invoked, which is difficult for native methods and expensive for Java library methods given the large number of reachable methods. Note that this treatment of Java library methods also assumes that any method invocation through the Java reflection API is assumed to invoke a method that has side effects.

Copernicus uses side-effect analysis to remove method invocations that target methods without side-effects, i.e., **hasSideEffects(m)** $= \mathtt{FALSE}$. Additionally, for methods invocations that return a value, that return value cannot be used. Such invocations often occur due to accessor methods in object-oriented frameworks, such as Ptolemy II.

The side effect analysis presented here is rather simple form of side-effect analysis targeted for elimination of dead-code method invocations. In particular, the analysis does not consider the presence of data reads and analyzes methods instead of individual statements within the method. This additional information could be used to perform additional optimizations, such as loop-invariant code motion, as described in [18].

6.5.3 Dead Field Analysis

A Java field is *dead* if its value is set, but never read. Dead field analysis computes a function **isDead** : $\mathit{fields} \to \mathbb{B}$. This function is defined by the least solution to the following constraints, computed on the lattice of boolean values $\mathbb{B}$ where $\mathtt{TRUE} \leq \mathtt{FALSE}$:

- Method m reads the value of field $f \implies \mathtt{FALSE} \leq \mathbf{isDead}(f)$

Static and non-static fields that are dead (**isDead(f)** $=$ `TRUE`) can be removed, along with any instructions that store to the field. As with other application extraction analyses, the implementation of dead field analysis is simplified by assuming that the public fields of generated code are not accessed by the standard Java libraries.

6.5.4 Obfuscation

Another optimization that occurs in step five is *obfuscation* of the generated code. Obfuscation replaces the names of all methods with shorter strings. This is important since, in Java bytecode, methods are referred to by the complete signature of the method. Hence, unlike statically linked C or C++ executables, the names of methods often have a significant impact on the size of compiled Java classes. Copernicus applies the obfuscator in Jode [41], an open source decompilation tool.

6.6 Performance

To analyze the performance of Copernicus, it was applied to 184 synchronous dataflow models. These models were originally created as tests for various features of Ptolemy II and utilize a wide variety of data types and library actors. Code was successfully generated for 130 models. The remaining 54 models use features of Ptolemy II that are not currently supported by Copernicus, such as record data types and programmatic traversal of a model. In most cases, the models were executed for 100000 iterations, corresponding to several seconds of simulation for the original Ptolemy II models on a 1.8 GHz Pentium M processor using Sun's JDK1.4.2_06. Due to large execution times, some models were executed for fewer iterations. Four models were not scalable to a reasonable execution time due to the design of the model or available input data and are not included in the performance data.

Figure 6.31 shows the speedup achieved Copernicus, comparing the execution time of generated code relative to the original simulation time. Models with low speedup generally use actors that limit the execution rate of the model, such as the Sleep actor. Other models with low speedup use coarse-grained actors, type-specific actors with relatively little communication (such as the FFT and RecursiveLattice actors). On the other hand, applying Copernicus to models with fine-grained actors, especially the Expression actor resulted in significant speedups. This data was collected with a large initial heap size in order to emphasize the speedup from reduced method indirection in the generated code, rather than the speedup from reduced load on the garbage collector.

Figures 6.32 and 6.33 shows the improvement in memory usage achieved using Copernicus. Figure 6.32 shows the *static* memory usage, consisting of memory allocated during execution of the model that cannot be garbage collected after execution. Static memory usage is a rough measure of object allocated during initialization, along with values that are computed and cached during execution. Figure 6.33 shows the *dynamic* memory usage, consisting of memory that is allocated and later freed by the Java garbage collector. Note that in some models that make use of arrays, it is possible for memory usage to increase (see section 6.4.3. In both cases, this data was collected by instrumenting the garbage collector using the `-Xloggc` command-line flag.

Figure 6.34 shows the size of the generated self-contained `.jar` file for each profiled model. The generated `.jar` file depends only on the standard Java libraries for execution. For comparison purposes, code generation with instrumentation for a minimal model requires 5156 bytes or 2734 bytes with obfuscation. Minimal `.jar` files for the original models were also created by *treeshaking* the models. Treeshaking involves executing the models using the `-verbose:class` command-line flag to report loaded classes and discarding any others. The minimal .jar files ranged from 710 to 770 Kilobytes. Note that the Matrix1 model, whose generated code requires over 700 Kilobytes, illustrates a case where residual dependencies on the Ptolemy II framework have not been removed by Copernicus. In this case, a method of the SDFDirector class is invoked directly by an actor in the model and not correctly removed by Copernicus, despite the fact that the code executes without error.

6.6.1 FIR Filter

A more in-depth example of specialized code generated by Copernicus is shown in Figure 6.35, which shows a simple model that applies a symmetric 31-tap filter to a simple generated signal. Ideally, execution of this model would roughly correspond to the hand-written code in Figure 6.36.

The main loop of the source code used for the FIR filter in Figure 6.37. The corresponding code generated from Copernicus is shown in Figure 6.38. This code was generated by decompiling Java bytecode generated by Copernicus using Jode [41], a Java decompiling utility. Although this code appears complex, the inner for loop is very similar to the hand-written inner loop. Much of the apparent complexity arises from the fact that a single Java statement can compile into several bytecode instructions and this structure cannot be easily recovered from the resulting code. Additionally, token unboxing results in many boolean flags representing whether variables reference an object or null. In this code, the flags happen to always be TRUE and are unnecessary, but this fact is not yet being leveraged by Copernicus. The performance of the generated code is shown in Figure 6.39.

For comparison, Figure 6.40 shows the C language filter code generated for a similar model constructed in Simulink, a commercial tool built by The Mathworks. This code stores the previous filter inputs in the Filter_STATE array and the filter taps in the Filter_C array. Although this code makes use of several C language features that are not available in Java, such as pointer arithmetic and implicit comparisons with zero, the structure of the basic loop is very close to the code generated by Copernicus.

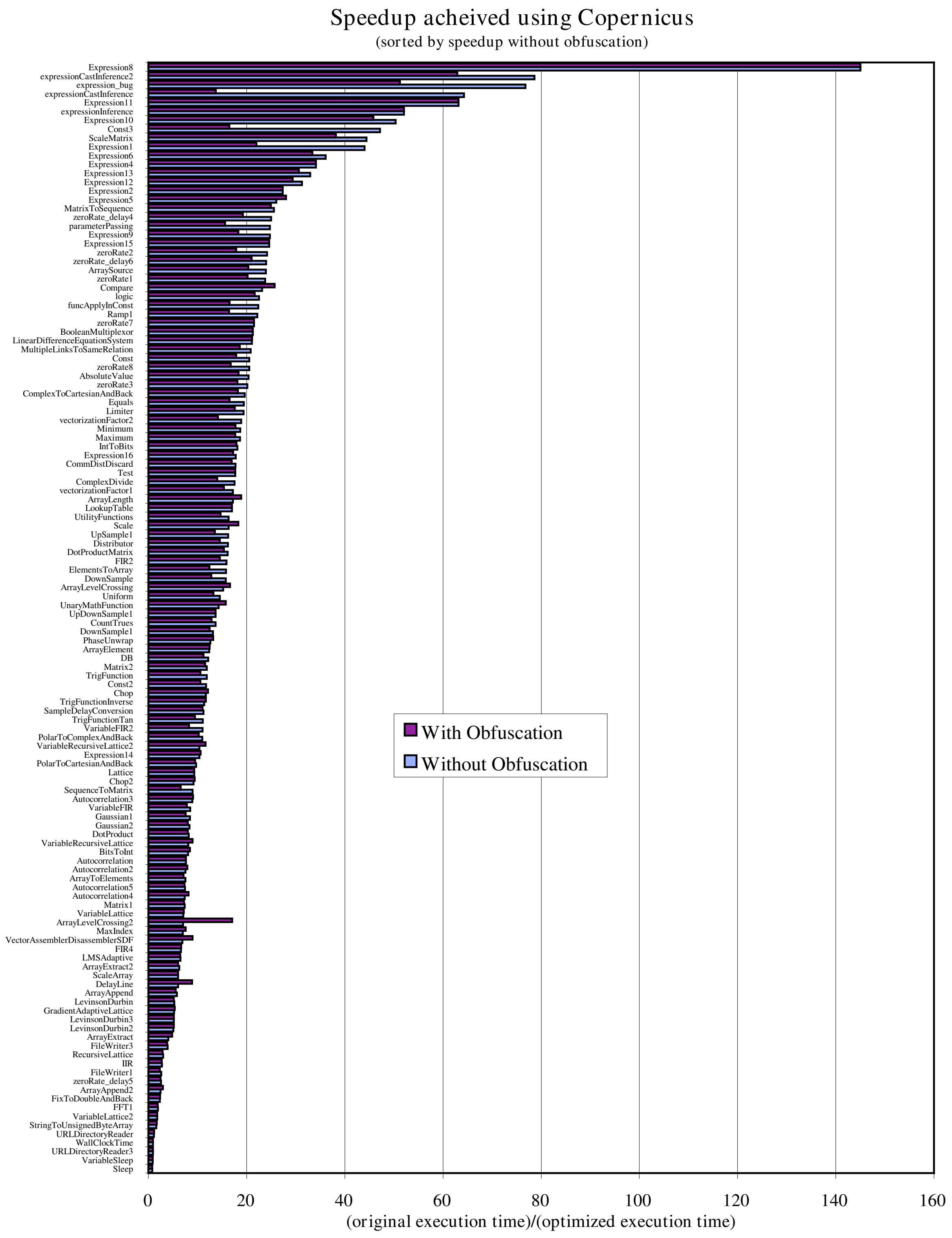


Figure 6.31: A graph of the speedup achieved by Copernicus, when applied to a number of test models.

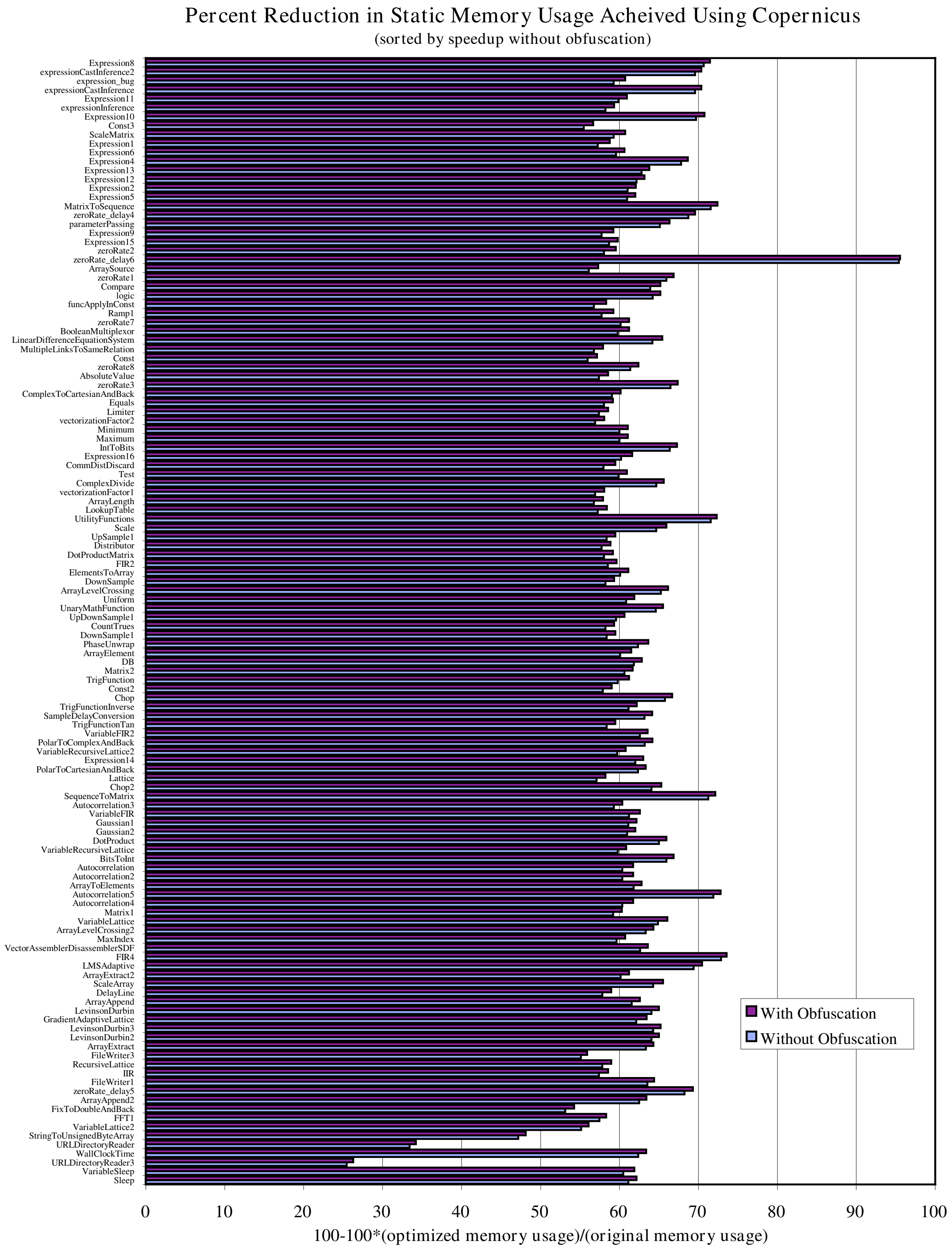


Figure 6.32: A graph of the reduction in static memory usage achieved by Copernicus, when applied to a number of test models.

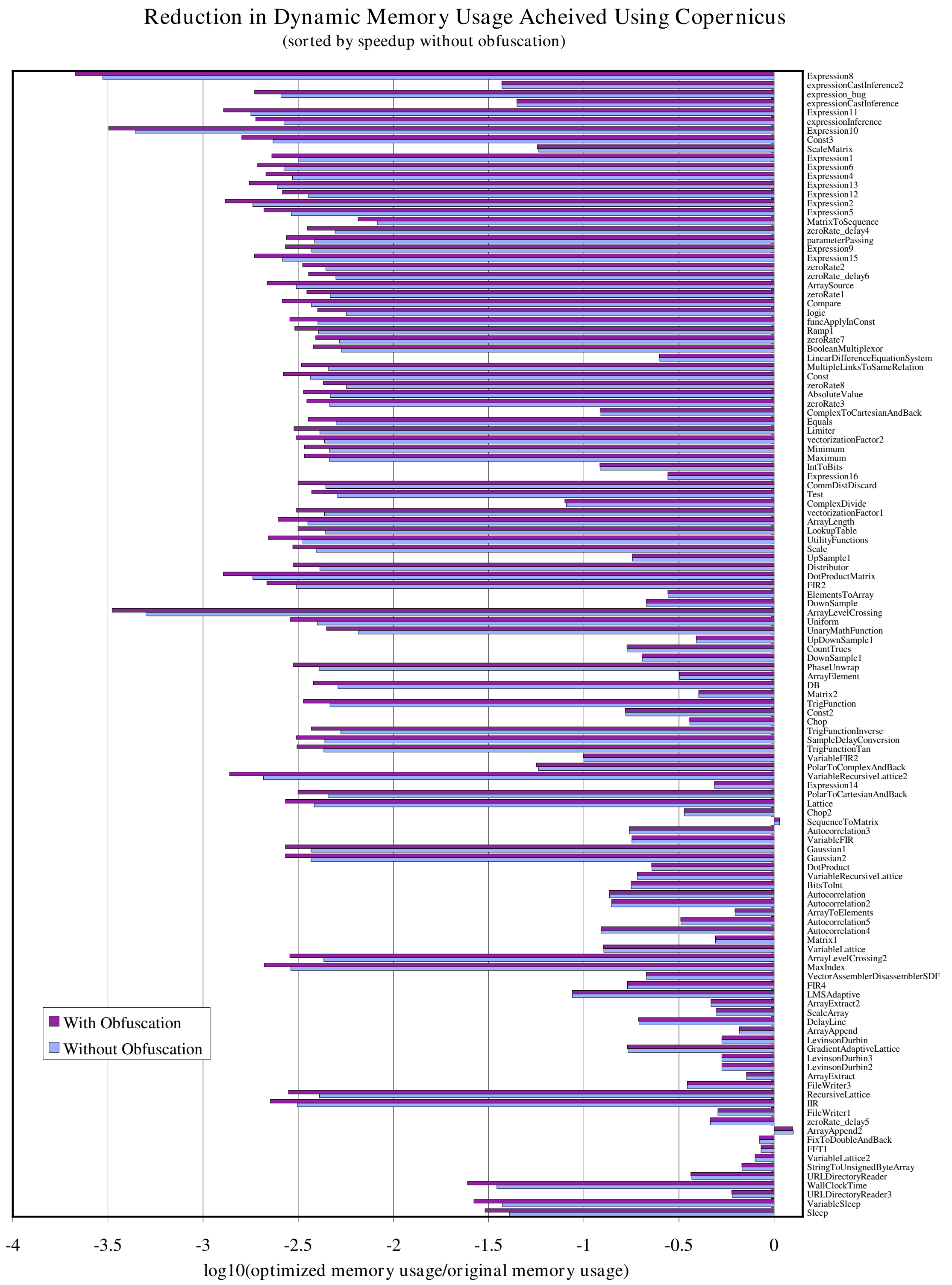


Figure 6.33: A graph of the reduction in dynamic memory usage achieved by Copernicus, when applied to a number of test models.

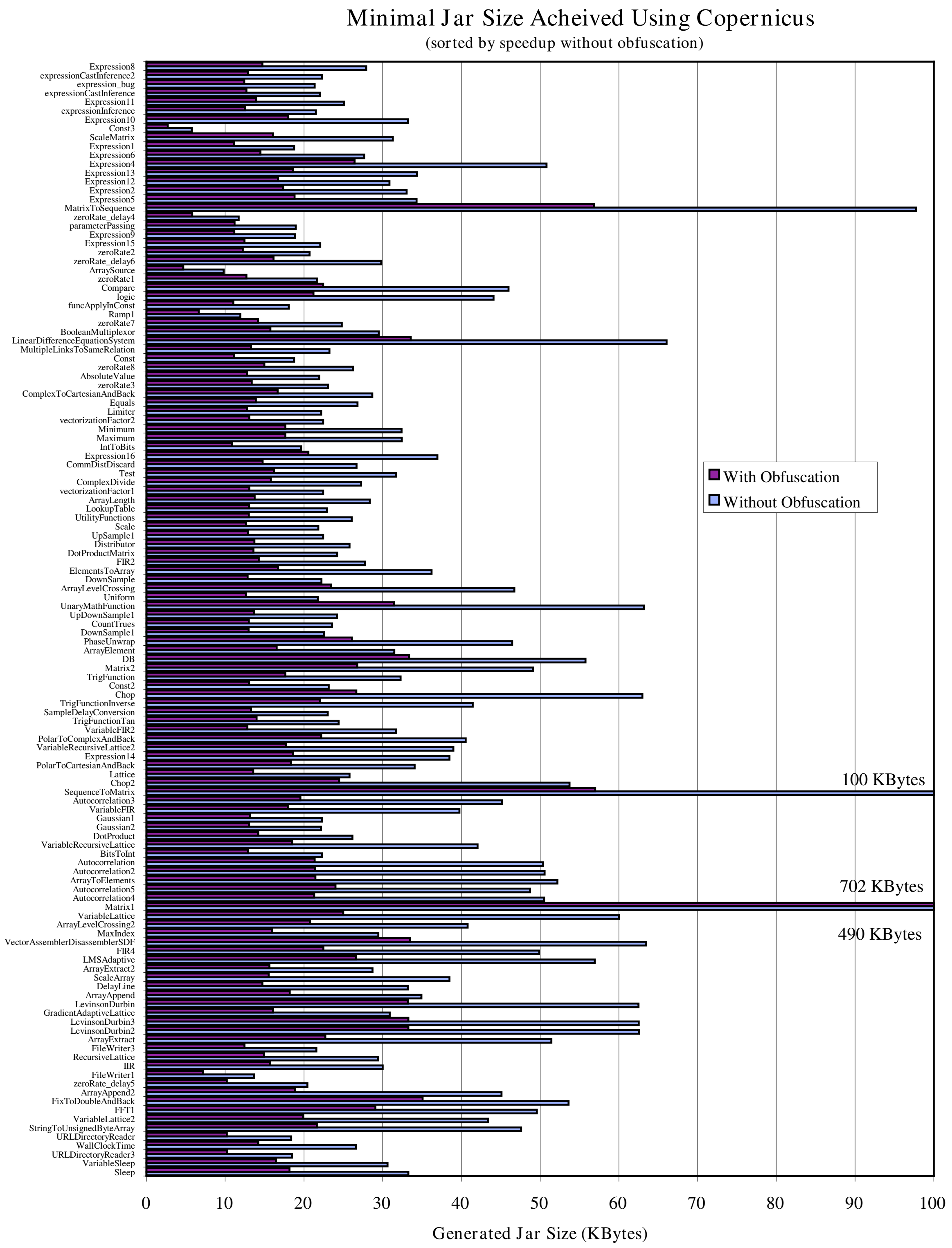


Figure 6.34: A graph of the minimal `.jar` size achieved by Copernicus, when applied to a number of test models.

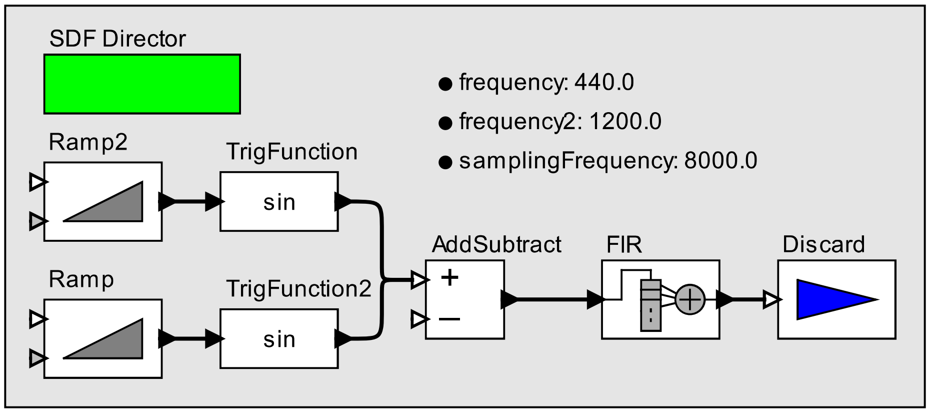


Figure 6.35: A model of an FIR filter system. For the purposes of profiling, the output is simply discarded.

6.6.2 Adaptive FIR Filter

Another example of how specialization can improve performance is shown in Figure 6.41. This model shows a simple model containing an Least Mean Square (LMS) Adaptive Filter, implemented by extending the Java class that defines the basic FIR filter. The same algorithm is shown implemented using a synchronous dataflow model in Figure 6.42. Ideally, there would be no performance difference between the code generated from the two different implementation of the LMSAdaptive actor.

Figure 6.43 compares the performance of the generated code for the two filter implementations. Copernicus reduces execution time and memory usage of both implementations immensely, although the hierarchically modeled filter remains somewhat slower than the filter implemented in Java code. The most significant difference between the two is the significant dynamic memory usage of the hierarchically modeled filter, even after applying Copernicus. This difference is because the modeled filter creates ArrayToken objects to represent the filter taps. Although Copernicus unboxes the ArrayToken objects, it does not eliminate the freshly allocated array contained by the ArrayToken.

```
public class FilterTest {
    static double _state_Ramp = 0.0;
    static double _state_Ramp2 = 0.0;
    static double[] _taps = new double[31];
    static double[] _delay = new double[31];
    static double _output = 0.0;
    public static void main( String[] args ) {
        /* Initialize */
        _state_Ramp = 0.0;
        _state_Ramp2 = 0.0;
        _taps[0] = 0.013689;
        // Remaining _taps initialization not shown.
        for ( int i = 0; i < 31; i++ ) { _delay[i] = 0; }
        for ( int i = 0; i < 800000; i++ ) {
            double sin1 = Math.sin( _state_Ramp );
            double sin2 = Math.sin( _state_Ramp2 );
            double sum = _sin1 + _sin2;
            _delay[0] = sum;
            for ( double d = 0.0, int j = 0; j < _taps.length; j++ ) {
                d += _taps[j] * _delay[j];
            }
            _output = d;
            /* State update. */
            _state_Ramp = _state_Ramp + 0.3455751918948773;
            _state_Ramp2 = _state_Ramp2 + 0.9424777960769379;
            System.arraycopy( _delay, 0, _delay, 1, _delay.length−1 );
        }
    }
}
```

Figure 6.36: Hand optimized FIR filter code.

```
public TypedIOPort input, output; // double
public Token[] _data;
public Token[] _taps;
public Token _zero;
    public void fire() throws IllegalActionException {
        super.fire();
        // Shift the delay line
        System.arraycopy( _data, 0, _data, 1, _data.length − 1 );

        _data[0] = input.get(0);
        Token outToken = _zero;
        for (int i = 0; i < _data.length; i++) {
            Token tapItem = _taps[i];
            Token dataItem = _data[i];
            dataItem = tapItem.multiply( dataItem );
            outToken = outToken.add( dataItem );
        }
        output.send(0, outToken);
    }
```

Figure 6.37: Original code from an actor representing a single-rate FIR filter.

```
public void fire() {
    boolean[] bools = _CG__data_isNotNull;
    double[] ds = _CG__data_value;
    int i = ds.length − 1;
    System.arraycopy(bools, 0, bools, 1, i);
    System.arraycopy(ds, 0, ds, 1, i);
    boolean[] bools_3_ = _CG__data_isNotNull;
    double[] ds_4_ = _CG__data_value;
    FilterTestModel filtertestmodel = _CGContainer;
    boolean[] bools_5_ = filtertestmodel._CG__relation3_0_double_isNotNull;
    double[] ds_6_ = filtertestmodel._CG__relation3_0_double_value;
    double d = ds_6_[0];
    bools_3_[0] = bools_5_[0];
    ds_4_[0] = d;
    boolean bool = _CG__zero_isNotNull;
    double d_7_ = _CG__zero_value;
    for (i = 0; i < ds_4_.length; i++) {
        bool = _CG__taps_isNotNull[i];
        double d_8_ = _CG__taps_value[i];
        bool = bools_3_[i];
        d = ds_4_[i];
        d_8_ *= d;
        d_8_ = d_7_ + d_8_;
        bool = true;
        d_7_ = d_8_;
    }
    bools_5_ = filtertestmodel._CG__relation4_0_double_isNotNull;
    double[] ds_9_ = filtertestmodel._CG__relation4_0_double_value;
    bools_5_[0] = bool;
    ds_9_[0] = d_7_;
}
```

Figure 6.38: Code generated by Copernicus by specialization of a generic single-rate FIR filter.

Figure 6.39: Performance comparison of FIR filter implementations. Profiling was taken over 800000 processed samples. The unusually long execution time for the obfuscated version of the handwritten code is apparently due to a bug in the Java just-in-time compiler.

	Code Size (bytes)	Execution Time (ms)	Memory Usage (Kbytes) Static	Memory Usage (Kbytes) Dynamic
Ptolemy II	743687	40217	324	1011032
Copernicus	23326	2590	112	189
with obfuscation	11237	2631	109	135
Handwritten Code	1770	1482	91	56
with obfuscation	1057	7031	87	38

```
int_T output;
{
    int_T nx = 30;
    real_T *x = &rtD.Filter_STATE[0];
    real_T *Cmtx = &rtP.Filter_C[0];
    while (nx−−) {
        output += (*Cmtx) * (*x++);
        Cmtx += 1;
    }
}
```

Figure 6.40: C code for an FIR filter generated from Simulink.

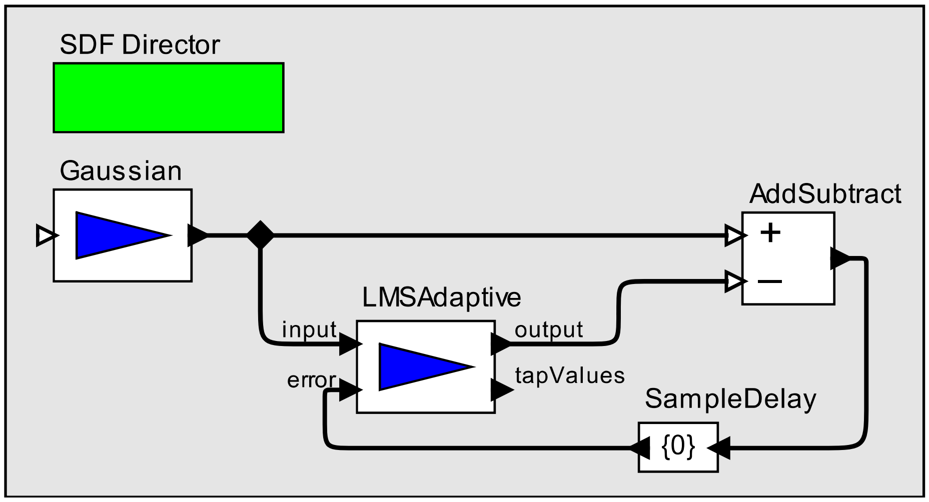


Figure 6.41: A model of an adaptive FIR filter system.

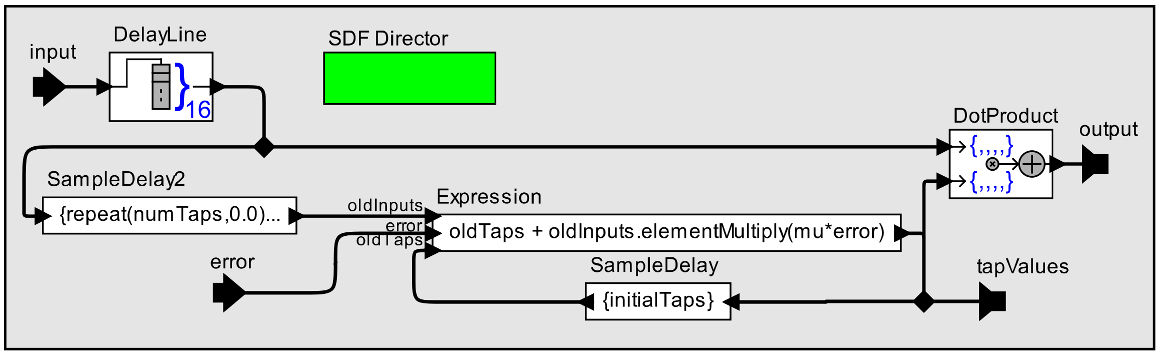


Figure 6.42: A model of an adaptive FIR filter.

Figure 6.43: Performance comparison of LMS adaptive filter implementations. Profiling was taken over 800000 processed samples.

	Code Size (bytes)	Execution Time (ms)	Memory Usage (Kbytes) Static	Memory Usage (Kbytes) Dynamic
Ptolemy II	738847	2103	326	195891
Copernicus	24434	297	113	151
with obfuscation	10753	290	110	106

(a) LMSAdaptive actor implemented in Java code

	Code Size (bytes)	Execution Time (ms)	Memory Usage (Kbytes) Static	Memory Usage (Kbytes) Dynamic
Ptolemy II	758073	6596	461	580523
Copernicus	47398	661	116	175244
with obfuscation	19150	664	113	175159

(b) LMSAdaptive actor implemented in a hierarchical model

Chapter 7

Conclusion

This thesis has addressed issues relating to the use of generic, parameterized actor-oriented components in hierarchical dataflow models. An actor-oriented component can be instantiated in different models, which allows for convenient design reuse and library-based design. Actor-oriented components can also be reconfigured during execution, which allows for many complex systems to be expressed easily. In many models, it becomes important to constrain how an actor can be reconfigured in a model. We treat constraints on reconfiguration as safety requirements which guarantee the absence of certain kinds of modeling errors. We have presented an analysis framework for verifying these reconfiguration constraints, which has been implemented in Ptolemy II. We have also leveraged information from the analysis in a metaprogramming system that transforms Ptolemy II models into self-contained Java code with improved performance.

Reconfiguration analysis is described in terms of a formalized mathematical framework. This framework abstracts the behavior of an actor-oriented model. One portion of this framework describes the dependencies between actor parameters and the effects of reconfiguration in the style of an attribute grammar. The remainder of the framework describes the sequencing constraints on the execution of an actor oriented model implied by the hierarchical structure of the model. This portion of the framework leverages the lattice structure of quiescent points in a model, implied by the fact that actor-oriented models are hierarchically reactive.

The reconfiguration analysis we have presented unifies many concepts in the dataflow literature. In particular, a source of reconfiguration, which we call a change context, is always associated with a single actor in a hierarchical model. This is true regardless of whether reconfiguration is specified using finite state machines, reconfiguration ports, or other syntaxes. The formal framework also unifies different types of safety constraints, allowing them to be considered in a generic way. These safety constraints may include both requirements that parameters do not change and requirements that parameters change only at particular points in the execution of a model. This second type of requirement allows verification of complex behavioral properties, such as the local synchrony constraint for parameterized synchronous dataflow scheduling.

The reconfiguration analysis depends on two conservative approximations in order to make the analysis problem tractable and efficient. Firstly, the theory analyzes the behavior of the model based on all possible executions of a model. If a reconfiguration constraint might be violated during any execution of the model, the theory assumes that the model is invalid. Secondly, the theory approximates the set of change contexts for a parameter by the least change context. The least change context approximation allows for efficient type checking, but might result in no information about reconfiguration. We show, however, that the least change context approximation is sufficient to check interesting semantic constraints. Because the reconfiguration analysis checks safety properties of an actor oriented model using efficient approximations, we call the analysis a behavioral type system.

In Ptolemy II, generic, parameterized actors are implemented in Java using an object-oriented framework. This framework supports reconfiguration at run-time using indirection. Unfortunately, each level of indirection introduces overhead into every operation. This overhead can be difficult to see, however, since it is hidden in the implementation and interaction of methods in different classes. To expose this overhead, we have developed a variation of UML class diagrams specifically to describe indirection in object-oriented frameworks such as Ptolemy II.

The thesis also describes a metaprogramming system called Copernicus that transforms Ptolemy II models into self-contained Java code. Copernicus analyzes an actor-oriented model to determine whether the context of an actor in the model changes due to reconfiguration. This context includes the structure of the model, the model of computation, data types, and parameter values. The Java code for each actor is then transformed to be specific to this context, allowing the indirection to be removed. Conceptually, a generic actor specification is specialized to a particular role in the model.

After specialization, the generated code is significantly more efficient than the original Ptolemy II model in terms of code size, memory usage, and execution speed. For small examples, this generated code approaches the resource usage of handwritten Java code. We anticipate that this efficient generated code, together with appropriate models of computation for specifying real-time behavior could eventually form a basis for embedded system implementation.

Appendix A

Mathematical Background

The mathematics required to understand this thesis is not complicated, however it may be unfamiliar to some readers. In the interest of self containment and notational clarity, this appendix will present a brief introduction to the mathematics of partially ordered sets as used in this thesis. Those readers interested in more detailed understanding are highly recommended to read Davey and Priestley [23], although trees are treated only in a brief exercise.

Notationally, elements in a set are usually written in lower case letters: a, sets are written in capital letters: A, and sets of sets are written in boldface capitals: $\mathbf{A}$. The set of natural numbers $\{0, 1, 2, ...\}$ and the set of booleans $\{\mathtt{TRUE}, \mathtt{FALSE}\}$ are written $\mathbb{N}$ and $\mathbb{B}$, respectively. The fact that an element a is contained in a set A is written: $a \in A$. The set $A \times B$, called the *product set* is the set of pairs of elements (a, b) where $a \in A$ and $b \in B$, which might also be written as set closure: $\{(a, b) : a \in A, b \in B\}$. A set A is a *subset* of another set B, written $A \subseteq B$, if every element of A is an element of B. This fact can also be written as $A \subseteq B \iff \forall a \in A, a \in B$. Note that A might contain exactly the same elements of B. If there is some element of B which is not contained in A, then A is a *strict subset* of B, written $A \subset B$. This fact can also be written:$A \subset B \iff \exists b \in B, b \notin A$. Given two sets A and B, their *union* is written $A \cup B$ and their *intersection* is written $A \cap B$.

A *relation between a set A and a set B*, is simply a subset of $A \times B$. For $a \in A$ and $b \in B$, the fact that r is in a relation R is equivalently written: $(a, b) \in R$ or $a\,R\,b$. The opposite fact is written: $(a, b) \notin R$. A relation might also be a subset of $A \times A$, in which case it is simply a *relation over A*. You are probably already familiar with the *identity relation over a set A*, written $=_A$, but you are probably used to ignoring the original set and just writing $=$. Many relations are distinguished in some way. A relation R over A is:

- *reflexive* if and only if $\forall a \in A, a\ R\ a$.
- *irreflexive* if and only if $\forall a \in A, (a, a) \notin R$.
- *symmetric* if and only if $\forall a_1 \in A, a_2 \in A, a_1\ R\ a_2 \iff a_2\ R\ a_1$.
- *antisymmetric* if and only if $\forall a_1 \in A, a_2 \in A, a_1\ R\ a_2 \wedge a_1\ R\ a_2 \implies a_1 = a_2$.
- *transitive* if and only if $\forall a_1 \in A, a_2 \in A, a_3 \in A, a_1\ R\ a_2 \wedge a_2\ R\ a_3 \implies a_1\ R\ a_3$.

The *reflexive closure* $R^?$ of a relation R over A is defined to be $R \bigcup =_A$. Note that the reflexive closure is reflexive by construction. The *transitive closure* R^+ of a relation R over A is defined to be smallest set that contains R, where $a_1\, R^+\, a_2 \wedge a_2\, R^+\, a_3 \implies a_1\, R^+\, a_3$. Note that the transitive closure is transitive by construction. The *reflexive, transitive closure* is $R^+ \bigcup =_A$, and is both reflexive and transitive.

Any transitive, irreflexive, antisymmetric relation over a set A can be interpreted as a *strict partial order* of elements, and is usually written with the familiar symbol: $<$. The reflexive closure of a strict partial order, often just called a partial order, is written with a similar symbol: $\leq$. A special kind of partial order where every element is related to every other element ($\forall a_1 \in A, a_2 \in A, a_1 \leq a_2 \vee a_1 \geq a_2$) is called a *total order*. Unfortunately, partial orders and total orders are usually written in the same way, so total orders will always be noted explicitly. A pair $(A, \leq_A)$ of a *carrier set* and a partial order over the set is called a *partially ordered set*, or simply a *poset*.

It is customary to draw simple posets in a *Hasse diagram*, where each element of the carrier set is represented by a dot and lines are drawn between comparable elements. Although in simple cases, the order is often inferred from the vertical position of two connected dots in a diagram, it is less confusing to indicate the order explicitly. Example of Hasse diagrams is shown in Figure A.1. [1]

[1]Incidentally, explicitly specifying the order is also consistent with a categorical interpretation of posets [84].

The *upset* of an element a is the set of all elements that are greater than a. Similarly, the *downset* of an element a is the set of all elements that are less than a. The *greatest lower bound* of a set A, written $\sqcap A$, is the unique element that is a lower bound for the set (i.e., is less than every element in the set) and also greater than every other lower bound. Conversely, the *least upper bound* of a set A, written $\sqcup A$, is the unique element that is an upper bound of the set and also less than every other upper bound. In an arbitrary order, the greatest lower bound and least upper bound of a set do not necessarily exist. A *lattice* is a special partial order where every subset of the lattice has a least upper bound and a greatest lower bound.

It is often important to consider posets with certain properties, or subsets of posets with certain properties. A *chain* is a simple name for a countable, totally ordered set. It is often useful to consider subsets of a poset which are also chains. Some partial orders have a top element, written $\top$, that is greater than every other element, or a bottom element, written $\bot$, that is less than every other element. A *top-rooted tree* is a poset with a top element, where the upset of every element is a chain. Symmetrically, a *bottom-rooted tree* is a poset with a bottom element, where the downset of every element is a chain. An example of a tree is shown in Figure A.1d.

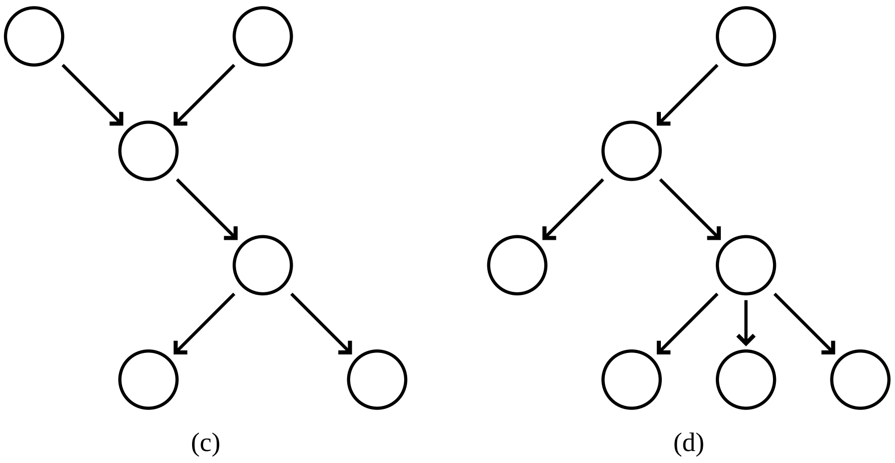


Figure A.1: Two diagrams illustrating partially ordered sets. The order is indicated with arrow between elements of the set, and redundant relations in the set are not shown explicitly. (c) is not a tree or a lattice, while (d) is a top-rooted tree.

Appendix B

Summary of Theorems

Definition 1 39

Consistent valuation function:

A valuation function v is consistent if and only if $\forall p \in \mathbf{P}$, p is dependent

$\implies constraint_p(v(domain_1^p), \ldots, v(domain_n^p)) = v(p)$

Definition 2 41

Constant parameter:

Parameter p is *constant* if and only if

$\forall a \in \mathbf{A}, \forall q \in Q^a, p \notin \overset{\rightsquigarrow}{R}(q)$.

Definition 3 41

Constant parameter over actor firings:

Parameter p is *constant over firings of actor c* if and only if

$\forall a \in \mathbf{A}, \forall q \in Q^a, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$.

Theorem 1 41

p is constant implies p is constant over firings of any actor.

Proof: Let c be an arbitrary element of $\mathbf{A}$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \notin \overset{\rightsquigarrow}{R}(q)$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$

p is constant over firings of c

Theorem 2 41

p is constant over firings of c and $c \trianglerighteq a$ implies p is constant over firings of a.

Proof: $\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$

$Q^c \subseteq Q^a$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^a$

p is constant over firings of a

Definition 4 42

Reconfiguration Requirement:

A *reconfiguration requirement* in a model m is a statement of the form "p is constant," or "p is constant over firings of actor a," where p and a are in the model.

Definition 5 42

Reconfiguration Safe:

An execution of a model with a set of reconfiguration requirements S is *reconfiguration safe* if the execution satisfies each requirement in S.

Definition 6 43

Change context:

An actor a is a change context of a parameter p, written $a \mapstochar\rightsquigarrow p$, if and only if $p \in \overset{\rightsquigarrow}{R}{}^a$.

Definition 7 43

Inherently constant parameter:

Parameter p is *inherently constant* if and only if

$\forall a \in \mathbf{A}, a \not\mapstochar\rightsquigarrow p$.

Definition 8 43

Inherently constant parameter over actor firings:

Parameter p is *inherently constant over firings of actor* a if and only if $\forall c \in \mathbf{A}, c \mapstochar$

$\rightsquigarrow p \implies c \trianglerighteq a$.

Theorem 3 43

p is inherently constant implies p is constant during any execution.

Proof: $\forall x \in \mathbf{A}, x \not\rightsquigarrow p$

$\forall x \in \mathbf{A}, p \notin \overset{\rightsquigarrow}{R}^x$

$\forall x \in \mathbf{A}, \forall q \in Q^x, \overset{\rightsquigarrow}{R}(q) \subseteq \overset{\rightsquigarrow}{R}^x$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \notin \overset{\rightsquigarrow}{R}(q)$

p is constant

Theorem 4 43

p is inherently constant over firings of actor c implies

p is constant over firings of actor c during any execution.

Proof: $\forall x \in \mathbf{A}, x \rightsquigarrow p \implies x \trianglerighteq c$

$\forall x \in \mathbf{A}, p \in \overset{\rightsquigarrow}{R}^x \implies x \trianglerighteq c$

$\forall x \in \mathbf{A}, \forall q \in Q^x, \overset{\rightsquigarrow}{R}(q) \subseteq \overset{\rightsquigarrow}{R}^x$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies x \trianglerighteq c$

$x \trianglerighteq c \implies Q^x \subseteq Q^c$

$\forall x \in \mathbf{A}, \forall q \in Q^x, p \in \overset{\rightsquigarrow}{R}(q) \implies q \in Q^c$

p is constant over firings of actor c

Definition 9 45

Least change context of a parameter:

The least change context of a parameter p, $\lfloor p \rfloor$, is an element of $\mathbf{A}_\perp^\top$ where $\lfloor p \rfloor = \sqcap\{a \in \mathbf{A}_\perp^\top : a \in \mathbf{A} \wedge a \rightsquigarrow p\}$

Or equivalently,

$$\lfloor p \rfloor = \begin{cases} \top & \text{if } \{a \in \mathbf{A} : a \rightsquigarrow p\} = \emptyset \\ \sqcap\{a \in \mathbf{A} : a \rightsquigarrow p\} & \text{if } \{a \in \mathbf{A} : a \rightsquigarrow p\} \neq \emptyset \text{ and} \\ & \sqcap\{a \in \mathbf{A} : a \rightsquigarrow p\} \text{ exists} \\ \perp & \text{otherwise} \end{cases}$$

Theorem 5 46

$\lfloor p \rfloor = \top$ implies p is inherently constant.

Proof: $\{a \in \mathbf{A} : a \mathrel{\mid\!\rightsquigarrow} p\} = \emptyset$

$\forall a \in \mathbf{A}, a \mathrel{\not\mid\!\rightsquigarrow} p$

p is inherently constant

Theorem 6 46

$\lfloor p \rfloor \in \mathbf{A}$ implies p is inherently constant over firings of $\lfloor p \rfloor$.

Proof: $\lfloor p \rfloor = \sqcap\{a \in \mathbf{A} : a \mathrel{\mid\!\rightsquigarrow} p\}$

$\forall a \in \mathbf{A} : a \mathrel{\mid\!\rightsquigarrow} p \implies a \trianglerighteq \lfloor p \rfloor$

p is inherently constant over firings of $\lfloor p \rfloor$

Theorem 7 47

p is inherently constant over $\mathit{actor}(p)$ implies that $\lfloor p \rfloor \neq \bot$.

Proof: By cases.

Let p be an arbitrary element of $\mathbf{P}$

Case 1: $\nexists c \in \mathbf{A}$ such that $c \rightsquigarrow p$

$\implies \lfloor p \rfloor = \top$

Case 2: $\exists$ unique $c \in \mathbf{A}$ such that $c \rightsquigarrow p$

$\implies \lfloor p \rfloor = c$

Case 3: $\exists A \subseteq \mathbf{A}$ such that $\forall c \in A, c \rightsquigarrow p$

$\forall c \in A, c \trianglerighteq \mathit{actor}(p)$

$(A, \trianglerighteq)$ is a chain

$\exists c \in A, \forall x \in A, x \trianglerighteq c$

$\implies \lfloor p \rfloor = c$

Theorem 8 47

$p_1 \rightsquigarrow p_2$ implies $\lfloor p_1 \rfloor \trianglerighteq_{\bot}^{\top} \lfloor p_2 \rfloor$.

Proof: Let p_1 and p_2 be arbitrary elements of $\mathbf{P}$

$$\forall a \in \mathbf{A}, a \mapsto\!\!\rightsquigarrow p_1 \implies a \mapsto\!\!\rightsquigarrow p_2$$
$$\{a \in \mathbf{A} : a \mapsto\!\!\rightsquigarrow p_1\} \subseteq \{a \in \mathbf{A} : a \mapsto\!\!\rightsquigarrow p_2\}$$
$$\{a \in \mathbf{A}_{\bot}^{\top} : a \in \mathbf{A} \wedge a \mapsto\!\!\rightsquigarrow p_1\} \subseteq \{a \in \mathbf{A}_{\bot}^{\top} : a \in \mathbf{A} \wedge a \mapsto\!\!\rightsquigarrow p_2\}$$
$$\sqcap\{a \in \mathbf{A}_{\bot}^{\top} : a \in \mathbf{A} \wedge a \mapsto\!\!\rightsquigarrow p_1\} \trianglerighteq_{\bot}^{\top} \sqcap\{a \in \mathbf{A}_{\bot}^{\top} : a \in \mathbf{A} \wedge a \mapsto\!\!\rightsquigarrow p_2\}$$
$$\lfloor p_1 \rfloor \trianglerighteq_{\bot}^{\top} \lfloor p_2 \rfloor$$

Theorem 9 47

$p \in R^c$ implies $c \trianglerighteq_{\bot}^{\top} \lfloor p \rfloor$.

Proof: Let c be an arbitrary element of $\mathbf{A}$ and p be an arbitrary element of R^c

$$p \in \overset{\rightsquigarrow}{R^c}$$
$$c \mapsto\!\!\rightsquigarrow p$$
$$c \in \{a \in \mathbf{A} : a \mapsto\!\!\rightsquigarrow p\}$$
$$c \in \{a \in \mathbf{A}_{\bot}^{\top} : a \mapsto\!\!\rightsquigarrow p\}$$
$$c \trianglerighteq_{\bot}^{\top} \sqcap\{a \in \mathbf{A}_{\bot}^{\top} : a \mapsto\!\!\rightsquigarrow p\}$$
$$c \trianglerighteq_{\bot}^{\top} \lfloor p \rfloor$$

Definition 10 49

Conditional Reconfiguration Function:

A *conditional reconfiguration function* $f_{a \rightsquigarrow p} : (\mathbf{P} \to \mathbf{V}) \times (\mathbf{P} \to \mathbf{A}_{\bot}^{\top}) \to \{a, \top\}$ for an actor a and a parameter p is monotonic function, where $f^a(v_0, \lfloor\cdot\rfloor) = a$ if in any execution beginning with parameter values v_0, $\forall q \in Q^a, p \in R(q)$.

Definition 11 49

Conditional Dependence Function:

A *conditional dependence function* $f_{p_1 \rightsquigarrow p_2} : (\mathbf{P} \to \mathbf{V}) \times (\mathbf{P} \to \mathbf{A}_{\bot}^{\top}) \to \mathbf{A}_{\bot}^{\top}$ for parameters p_1 and p_2 is monotonic function, where $f_{p_1 \rightsquigarrow p_2}(v_0, \lfloor\cdot\rfloor) = \lfloor p_1 \rfloor$ if in any execution beginning with parameter values v_0, reconfiguration of p_1 requires evaluation of $\mathit{constraint}_{p_2}$

Bibliography

[1] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts : A tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

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

[4] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag, May 2003.

[5] Lennart Augustsson, Jacob Schwartz, and Rishiyur Nikhil. Bluespec language definition. Technical report, Sandburst Corporation, November 2000.

[6] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of java without data races. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 382–400. SIGPLAN, ACM Press, 2000.

[7] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In A. Chalmers, M. Mirmehdi, and H. Muller, editors, *Proceedings of the Conference on Communicating Process Architectures*, pages 1–14. IOS Press, September 2001.

[8] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Quasi-static scheduling of

reconfigurable dataflow graphs for DSP systems. In *Proceedings of the International Workshop on Rapid System Prototyping (RSP)*. IEEE, June 2000.

[9] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[10] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Consistency analysis of reconfigurable dataflow specifications. In *Embedded Processor Design Challenges*, number 2268 in Lecture Notes in Computer Science, pages 1–17. Springer-Verlag, October 2002.

[11] Shuvra S. Bhattacharyya, Pravin K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[12] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[13] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *The Journal of Supercomputing*, 21(2):117–130, 2002.

[14] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *International Journal of Computer Simulation*, 4:155–182, April 1994. Special issue on "Simulation Software Development".

[15] Joseph Buck and Radhu Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, May 2000.

[16] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1993.

[17] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the Asilomar Conference on Circuits, Signals and Systems*. IEEE, October 1994.

[18] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, November 1997.

[19] M. Cole and S. Parker. Dynamic compilation of C++ template code. *Scientific Programming*, 11:321–327, 2003.

[20] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Symposium on Principles of Programming languages (POPL)*. SIGPLAN, ACM, January 1993.

[21] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, tools, and applications*. Addison-Wesley, May 2000.

[22] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard.* Springer-Verlag, 2002.

[23] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[24] Jack B. Dennis. First version of a dataflow procedure language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science, pages 362–376. Springer-Verlag, April 1974.

[25] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*. Number 323 in Lecture Notes in Computer Science. Springer-Verlag, 1988.

[26] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 7–17. SIGPLAN, ACM, 1997.

[27] Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357. SIGPLAN, ACM, 2000.

[28] B. Draper, W. Böhm, J. Hammes, W. Najjar, R. Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins. Compiling SA-C programs to FPGAs: Performance results. In *Proceedings of the International Conference on Vision Systems (ICVS)*, number 2095 in Lecture Notes in Computer Science, pages 220–235. Springer-Verlag, July 2001.

[29] Johan Eker et al. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1), January 2003.

[30] Johan Eker and Jörn Janneck. CAL language report: Specification of the CAL actor language. Technical Memorandum UCB/ERL 03/48, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, December 2003.

[31] Daniel D. Gajski, editor. *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[33] G.R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages V.561–564, March 1992.

[34] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[35] Aleksey Gurtovoy and David Abrahams. The Boost C++ metaprogramming library. Available at `www.boost.com`, March 2002.

[36] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[37] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on software engineering and methodology*, 5(4), 1996.

[38] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977.

[39] Carl Hewitt and Henry Baker. Actors and continuous functionals. In *Proceeding of Working Conference on Formal Description of Programming Concepts*, pages 267–387. International Federation for Information Processing, August 1977.

[40] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of the IFIP Conference on Very Large Scale Integration*, pages 595–619. International Federation for Information Processing, Kluwer, December 1999.

[41] Jochen Hoenicke. Jode: Java optimizer and decompiler. http://jode.sourceforge.net.

[42] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the International Conference on Software Reuse*, pages 134–142. IEEE, June 1998.

[43] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.

[44] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Technical Memorandum M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, March 2001.

[45] Jörn W. Janneck. Actors and their composition. *Formal Aspects of Computing*, 15(4):349–369, 2003.

[46] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, June 1993.

[47] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, pages 471–475. International Federation for Information Processing, North-Holland, 1974.

[48] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress*, pages 993–998. North-Holland, 1977.

[49] Asawaree Kalavade. *System-Level Codesign Of Mixed Hardware-Software Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.

[50] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: Run-time code generation for Java and its applications. In *International Symposium on Code Generation and Optimization (CGO)*, pages 48–58. IEEE, March 2003.

[51] D. J. Kaplan. An introduction to the processing graph method. In *Proceedings of the International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS)*, pages 46–52, March 1997.

[52] Holger Keding, Martin Coors, Olaf Luethje, and Heinrich Meyr. Fast bit-true simulation. In *Proceedings of the Design Automation Conference (DAC)*, June 2001.

[53] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[54] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 327–353, 2001.

[55] Gregor Kiczales, John Lampinga, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, 1997.

[56] Bart Kienhuis and Ed F. Deprettere. Modeling stream-based applications using the SBF model of computation. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 34(3):291–300, 2003.

[57] S. C. Kleene. *Introduction to Metamathematics*, chapter 12, page 342. North-Holland, 1952.

[58] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[59] Eddie Kohler, Robert Morris, and Benjie 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.

[60] Peeter Laud. Analysis for object inlining in java. In Uwe Aßmann, editor, *Java Optimization Strategies for Embedded Systems (JOSES) Workshop, in association with European Joint Conferences on Theory and Practice of Software (ETAPS)*, number 2001-10 in Technical Report, Genova, Italy, April 2001. Fakultät für Informatik, University of Karlsruhe.

[61] Bilung Lee. *Specification and Design of Reactive Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2000.

[62] Edward A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[63] Edward A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[64] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1):24–35, January 1987.

[65] Edward A. Lee and Stephen Neuendorffer. Actor-oriented models for codesign. In Sandeep Shukla and Jean-Pierre Talpin, editors, *Formal Methods and Models for System Design*. Kluwer, 2004. to appear.

[66] Edward A. Lee and Stephen Neuendorffer. A survey of embedded system models of computation. *Proceedings on Computers and Digital Techniques special issue on Embedded System Design*, 2004. to appear.

[67] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, June 2003.

[68] Edward A. Lee and Steve Neuendorffer. MoML - a modeling markup language in XML Version 0.4. Technical Memorandum UCB/ERL M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, March 2000.

[69] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[70] X. Leroy. Effectiveness of type-based unboxing. Technical Report BCCS-97-03, Boston College Computer Science Department, June 1997. In Workshop on Types in Compilation '97.

[71] Stan Y. Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the Design Automation Conference (DAC)*. SIGDA, ACM, 1997.

[72] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee. A hierarchical hybrid system model and its simulation. In *Proceedings of the Conference on Decision and Control (CDC)*, December 1999.

[73] Jie Liu. *Responsible Frameworks for Heterogenous Modeling and Design of Embedded Systems*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2001.

[74] B. Ludaescher, I. Altintas, and A. Gupta. Compiling abstract scientific workflows into web service workflows. In *Proc. of the Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, 2003.

[75] M. D. McIlroy. Mass-produced software components. In *Nato Science Committee Meeting at Garmisch, Germany*, October 1969.

[76] Walcelio Melo, Lionel Briand, and Victor Basili. Measuring the impact of reuse on software quality and productivity. Technical Report CS-TR-3395, University of Maryland, College Park MD, USA 20742, January 1995.

[77] Anne-Francoise Le Meur and Charles Consel. Generic software component configuration via partial evaluation. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, August 2000.

[78] John C. Mitchell. Coercion and type inference. In *Proceedings of the Symposium on Principles of Programming languages (POPL)*. ACM, January 1984.

[79] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[80] Praveen K. Murthy, Etan G. Cohen, and Steve Rowland. System Canvas: A new design environment for embedded DSP and telecommunication systems. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*. SIGDA, ACM, April 2001.

[81] Stephen Neuendorffer. Automatic specialization of actor-oriented models in Ptolemy II. Technical Memorandum M02/41, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, December 2002.

[82] Rishiyur Nikhil. Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications. In *Proceedings of the Conference on Methods and Models for Codesign (MEMOCODE)*, June 2004.

[83] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Department, University of California at Berkeley, CA, 1995.

[84] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[85] Jose Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck. Software synthesis for DSP using Ptolemy. *Journal on VLSI Signal Processing*, 9(1):7–21, January 1995.

[86] H. John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, School of Electrical Engineering, University of Technology, Sydney, 1995.

[87] Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.

[88] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Proceedings of the Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 211–258. Springer-Verlag, January 1980.

[89] Ulrik 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.

[90] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, number 1628 in Lecture Notes in Computer Science, pages 367–390. Springer-Verlag, June 1999.

[91] Ulrik P. Shultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4):452–499, July 2003.

[92] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[93] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[94] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. Technical Report 21451(96813), IBM Research, October 1999.

[95] Jeff Tsay, Christopher Hylands, and Edward Lee. A code generation framework for Java component-based designs. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems(CASES)*, pages 18–25, ACM, November 2000.

[96] Rob van Ommerling. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[97] Todd Veldhuizen. C++ templates as partial evaluation. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999.

[98] Piet Wauters, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-dynamic dataflow. In *Workshop on Parallel and Distributed Processing (PDP)*. EUROMICRO, IEEE, January 1996.

[99] Darren Webb, Andrew Wendelborn, and Kevin Maciunas. Process networks as a high-level notation for metacomputing. In *Proc. of the Int. Parallel Programming Symposium (IPPS), Workshop on Java for Distributed Computing*, 1999.

[100] Lars Wernli. Design and implementation of a code generator for the CAL actor language. Technical Memorandum UCB/ERL M02/5, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California Berkeley, CA 94720, USA, March 2002.

[101] Edward D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.

[102] Michael Winter et al. Components for embedded software : The PECOS approach. In *Workshop on Composition Languages, In conjunction with the European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[103] Yuhong Xiong. *An Extensible Type System for Component-Based Design*. PhD thesis, EECS Department, University of California at Berkeley, CA, 2002.