Hardware Synthesis of Synchronous Data Flow Models

Matthew R. Koecher
Brigham Young University - Provo

HARDWARE SYNTHESIS OF SYNCHRONOUS DATA FLOW MODELS

by

Matthew R. Koecher

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Brigham Young University

December 2003

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Matthew R. Koecher

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Michael J. Wirthlin, Chair
Date	Brent E. Nelson
Date	Doran K. Wilde

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Matthew R. Koecher in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Michael J. Wirthlin
Chair, Graduate Committee

Accepted for the Department

Michael A. Jensen
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, College of Engineering and Technology

ABSTRACT

HARDWARE SYNTHESIS OF SYNCHRONOUS DATA FLOW MODELS

Matthew R. Koecher

Department of Electrical and Computer Engineering

Master of Science

Synchronous Dataflow (SDF) graphs are a convenient way to represent many signal processing and dataflow operations. Nodes within SDF graphs represent computation while arcs represent dependencies between nodes. Using a graph representation, SDF graphs formally specify a dataflow algorithm without any assumptions on the final implementation. This allows an SDF model to be synthesized into a variety of implementation techniques including both software and hardware.

This thesis presents a technique for generating an abstract hardware representation from SDF models. The techniques presented here operate on SDF models defined structurally within the Ptolemy modeling environment. The behavior of the nodes within Ptolemy SDF models is specified in software and can be simple, such as a single arithmetic operation, or arbitrarily complex. This thesis presents a technique for extracting the behavior of a limited class of SDF nodes defined in software and generating a structural description of the SDF model based on primitive arithmetic and logical operations. This synthesized graph can be used for subsequent hardware synthesis transformations.

ACKNOWLEDGMENTS

This thesis is the culmination of the efforts of many people. To all those who have helped through encouragement or advice, I would like to express my sincere appreciation. Some special recognition should be given to Dr. Wirthlin, my thesis adviser. His advice and direction during the entire process helped guide me in my research. Most recently, his tireless efforts in helping me to revise have dramatically improved the quality of this thesis. Without his help this thesis could not have been realized. I would also like to express gratitude to my wife, Aubrey. Often working on this thesis caused me to be away, but she always encouraged me to finish what I had started. Her love and support when I felt frustrated were key to helping me complete this work. Finally, to all family, friends and committee members I owe my thanks. Their encouragement and support gave me the strength to keep working until I had finished.

Contents

List of Figures

Chapter 1

Introduction

Dataflow computing is a unique computing approach designed to exploit the natural concurrency found in many application-specific operations[1]. Rather than specifying the behavior of a system sequentially using a conventional imperative language, the behavior of dataflow computing is defined by specifying the operations and their data dependencies[2]. By defining computations graphically through the data-dependencies of operations, the natural parallelism of the computation can be easily identified and exploited. Dataflow programming is a natural way of defining many signal processing algorithms and other data-intensive computations.

The dataflow model of computation does not place any restrictions on the computing functions allowed within nodes of a dataflow graph. Perhaps the only restriction placed on dataflow computing nodes is that they must represent a computable operation. This flexibility suggests that the complexity of the computing functions performed by dataflow nodes may vary greatly. Dataflow nodes can be as simple as basic arithmetic operations or as complex as a complete signal-processing algorithm.

While this flexibility allows the creation of arbitrarily complex dataflow models, it poses a challenge to compilers and synthesis tools aimed at generating an efficient implementation of a dataflow model. Most tools that generate a target implementation from a dataflow graph place limits on the types of computations that may be found within the graph. This finite set of allowable operations usually corresponds to the set of operations supported by the target architecture. Each allowable operation within the predefined node library corresponds to an optimized implementation of the operation for the target architecture.

One form of dataflow that is particularly appealing for software code generation and hardware synthesis is called *Synchronous Data Flow* or SDF[3]. The key difference between *synchronous* dataflow and general dataflow models is the use of statically specified data consumption and production rates on SDF nodes. The static nature of data production and consumption rates facilitates compile-time analysis and eliminates the need for expensive run-time control. Static analysis of SDF models has been incorporated into many software and hardware synthesis tools.

The work presented in this thesis provides the ability to *synthesize* an abstract graph for a limited class of arbitrarily complex operations defined in software. The technique presented in this thesis analyzes the behavior of arbitrary SDF operations created for the Ptolemy II modeling environment. The behavior of these SDF actors is described using the Java programming language and conforms to the interface restrictions imposed by the Ptolemy II modeling environment. The behavior of these actors is analyzed and converted into an abstract hardware graph based on primitive operations. This synthesized graph can then be used in subsequent implementation-specific processing steps such as software code-generation or hardware synthesis.

1.1 Synthesis Strategy

The goal of this work is to synthesize a representation for arbitrary SDF models that can be readily transformed into hardware. This work does not attempt to actually create hardware. Instead, an intermediate hardware representation is created that can be targeted to a number of hardware architectures including FPGAs or an ASIC library. For this reason, the final output will be a representation called an abstract circuit graph. This graph will closely model a hardware circuit, and can be easily transformed into a structural circuit description with any number of hardware description languages, such as VHDL or Verilog.

The synthesis strategy described in this work will generate an abstract hardware circuit that executes a complete iteration of an SDF model in a single clock cycle. This approach will result in a hardware circuit that provides a dedicated resource for each hardware operation in the SDF graph. While other techniques are possible, this approach ensures the greatest levels of concurrency by executing every operation in parallel. The throughput of circuits generated in this way can be improved by employing common retiming and pipelining techniques[4, 5].

This synthesis strategy involves two major tasks. The first task is to extract the behavior from individual actors within the SDF graph. This task involves the analysis of Java code and generates a dataflow graph representation of the Java behavior. The second task involves the composition of actor dataflow graphs according to the topology of the original SDF specification. The end result of this process is a graph representing the behavior of the entire SDF model composed of primitive operations.

1.2 Related Work

Other projects have undertaken to synthesize hardware from Java code. These include a JHDL synthesis project[6], the Forge project from Xilinx, and SeaCucumber[7]. Whereas this work synthesizes code from programs written for Ptolemy semantics, the work in Worth[6] synthesizes hardware from Java programs written to JHDL semantics. JHDL[8] is a hardware description language written in Java. Worth takes a behavioral JHDL description and, through an analysis of Java bytecodes, produces a structural JHDL description. JHDL tools can synthesize the structural JHDL description into hardware.

The SeaCucumber[7] (SC) project synthesizes Java code to FPGA hardware. SC exploits coarse-grained parallelism by synthesizing programs that use concurrent Java threads. SC can also extract fine-grained parallelism by employing standard compiler techniques on each individual thread. SC and the research described here are similar in their approach to exposing fine-grained parallelism using control flow and dataflow analysis, but the coarse-grained parallelism is exposed in this work through the SDF actors.

The Forge project at Xilinx synthesizes Java code to a Verilog description. While the work described in this thesis synthesizes Java code that is written to the Ptolemy II design specifications, the Forge project seeks to synthesize arbitrary Java code. The goal of the Forge project to enable *architectural synthesis*, where the hardware is described at a high-level, architectural view. The Forge tool will synthesize this view to FPGA hardware.

1.3 Thesis Contribution

Several research projects have investigated novel ways for synthesizing hardware from SDF models. Each of these projects rely on SDF graphs composed of nodes selected from a pre-defined hardware library. The primary tasks of these synthesis techniques are the scheduling of hardware execution units and the optimization of communication buffers. None of these projects provide the ability to synthesize hardware from arbitrary actors described in software.

The primary contribution of this thesis is the description of a process to generate a hardware description from data flow operations described in software. Key contributions which aid in this process are the interval analysis algorithm, and the conversion of SDF graphs into homogeneous abstract circuit graphs. Interval analysis is the method for identifying parallelism in the Java bytecode of the SDF actors, and it is described in Chapter 3. SDF graphs as specified do not closely match hardware semantics, so this thesis demonstrates a method for converting the SDF graphs into a more convenient representation for hardware called the abstract circuit graph. This description is straightforward to translate to a desired hardware target. This transformation process is described in Chapter 4. Together, these techniques can be used to synthesize an SDF graph composed of software actors to hardware.

1.4 Thesis Organization

The synthesis techniques described in this thesis are limited to the synchronous dataflow model of computation. Chapter 2 will introduce SDF and provide several examples. Several efforts at synthesizing SDF implementations will also be described. This chapter will also introduce the Ptolemy modeling framework used by this work.

One of most important tasks of this synthesis process is the extraction of behavior from arbitrary Ptolemy actors defined in software. Chapter 3 will describe how Java bytecodes are converted into dataflow representations. A simple FIR filter example will be used to demonstrate this approach.

The second major task is to combine the synthesized actors into a single abstract hardware representation. This will be described in detail in Chapter 4. This composition is guided by the topology of the original SDF model. An important step in this process is the conversion of multi-rate SDF graphs into a single-rate homogeneous graph. Chapter 5 will provide a conclusion and discuss directions for future work.

Chapter 2

Synchronous Data Flow

The hardware synthesis technique presented in this thesis is based on the synchronous dataflow (SDF)[3] model of computation. SDF is used in many algorithms that are dominated by dataflow, particularly digital communication and filter applications[9]. An advantage of SDF is that it reveals the natural parallelism in an algorithm, as the only dependencies in an SDF graph are actual data dependencies. Because of its static nature and lack of control flow, SDF is a good specification approach for both software and hardware. This chapter will provide a background description of SDF and discuss its usefulness as a specification format for software and hardware synthesis.

This chapter will begin by defining dataflow graphs and synchronous dataflow in particular. Much of the terminology used in SDF literature will be introduced here. Scheduling of SDF graphs will then be discussed, including why scheduling is important, and the impact scheduling can have on SDF implementation. The chapter will then discuss previous implementations approaches for SDF. These include several software code generation and hardware synthesis examples. This chapter will conclude by introducing the Ptolemy modeling environment. The Ptolemy modeling environment provides a software implementation of SDF and will serve as the specification environment for the SDF models synthesized in this approach.

2.1 Synchronous Dataflow Graphs

Synchronous data flow or SDF is a well-known formalism or model of computation based on the dataflow computing concept[3]. A dataflow graph is defined as a directed graph with nodes (also called actors) that represent computations. Actors are self-contained computation units with no restrictions on the complexity of an actor's computation. It could be as simple as a single logic or arithmetic operation. It could also be a full DSP filtering algorithm, or even more complex. Figure 2.1 demonstrates a simple synchronous dataflow model with three actors.

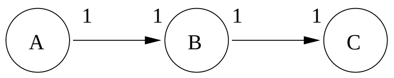

Figure 2.1: Simple synchronous dataflow model.

A dataflow graph also contains directed arcs which represent communication channels between the actors. All communication between actors in a SDF model is represented by an arc in the dataflow graph. The use of explicit arcs for actor communication prevents "side-effects" or implicit communication from one actor to another. In the SDF model of Figure 2.1, a communication arc is specified between actor A and B and between actor B and C. There is no direct communication between actor A and actor C.

The data takes the form of "samples" or "tokens" that are transmitted from the output of one node to the input node on the other end of the arc. A token is an encapsulation of a data value which can be of any format (such as integer, string, floating point, complex structured types, etc.). A token is *produced* when an actor sends a token on its output arc. A token is *consumed* when an actor accepts a token on one of its input arcs. The communication channels between actors have first-in, first-out (FIFO) queuing semantics. When a token is produced, it sits in the FIFO queue represented by the communication arc until the actor on the other side consumes it.

The primitive unit of computation in an SDF model is the "firing" of an actor. The firing of an actor involves three specific steps. First, the actor consumes tokens queued on its input arcs. These tokens were placed in the queue by the previous firing of the source actor of the input arc. Next, the actor performs the actor-specific computation using the data it received from the input arcs and state saved from previous actor firings. As described earlier, the actor computation may be any arbitrary operation. Finally, the actor produces new tokens on its output arcs based on the results of the actor-specific computation.

2.1.1 Port Token Rates

Each actor contains ports that provide an interface between the actor and SDF arcs. Arcs in an SDF model connect the output port of one actor to the input port of another actor. During the execution of a single actor firing, each actor port consumes or produces a fixed number of tokens. The number of tokens an actor's input port consumes or an output port produces during a firing is called the port's token rate. In the example of Figure 2.1, actor B consumes one token on its input port and produces one token on its output port during each firing.

It is not required that the token rates of two connected ports be equal. Figure 2.2 shows an example where the output port of actor A has a token rate of 2, while the input port of actor B has a token rate of 3. Actor A must fire more often than actor B in order to ensure that actor B has an adequate number of tokens to fire. Specifically, actor A must fire three times for every two firings of actor B.

An SDF graph is said to be *homogeneous* if the production and consumption rate on each port is one. The SDF graph of Figure 2.1 is an example of a homogeneous SDF graph. If the production or consumption rate of *any* port is more than one, the

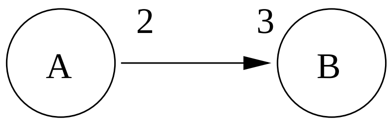

Figure 2.2: Example of ports with unequal token rates.

graph is referred to as *multi-rate*. The example of Figure 2.2 demonstrates a multi-rate SDF graph.

2.1.2 Initial Tokens

Another property of SDF graphs is that the arcs can contain initial tokens, or delays. If an arc has an initial token, then the receiving node will read that initial token before it will read the token output by the sending node. Initial tokens are indicated on an arc by a number followed by a D (for *delay*). Figure 2.3 demonstrates a simple SDF model that contains an initial token. In this example, the arc between actor A and B is annotated with a single sample delay (i.e., "1D"). When actor B first fires, it consumes the initial token on the delay arc. The token produced during the first firing of actor A will be consumed during the second firing of actor B.

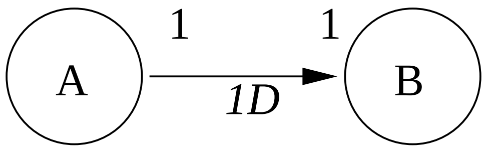

Figure 2.3: SDF example containing an initial token.

Initial tokens on arcs are effectively delays within the SDF graph. The nth token produced by an actor will be the $(n + m)$th token read by the input actor, where m is the number of initial tokens on the arc. Delays on an arc can affect the precedence relationship between actors. In the example of Figure 2.3, actor B can fire *before* actor A. However, in order to fire a second time, actor A will have to fire at least once.

SDF graphs may contain feedback. However, any feedback loop must contain initial tokens to avoid deadlock. Figure 2.4 shows an SDF graph with feedback. The initial token on the arc between actor D and actor A allows actor A to fire before any other actor in the graph. Actor A then produces a token allowing actor B to fire. This process continues and allows actor C and D to fire in turn. Without this initial token, no actor in the graph has sufficient tokens on its inputs to fire. This situation is called deadlock.

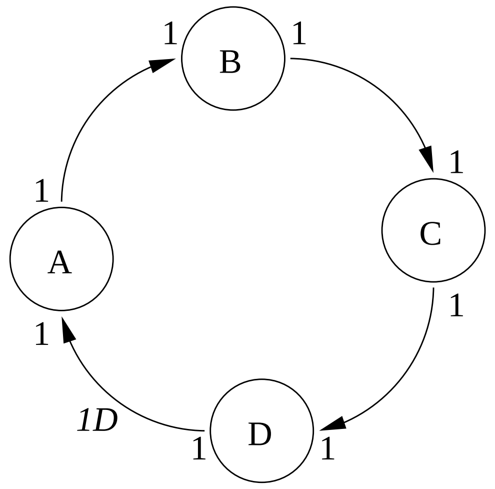

Figure 2.4: SDF graph with feedback and initial tokens.

2.2 Scheduling SDF

In any implementation of an SDF model a schedule must be created that orders the firing of each actor in the model. The firing rules of an SDF model require that an actor have sufficient tokens on all of its inputs before the actor can be fired. This SDF schedule must fire the actors in an order that ensures each actor has sufficient tokens in its input ports.

A schedule of an SDF model can be defined by a finite sequence of actor firings called an *iteration*. During the execution of an iteration, the graph must be deadlock free and ensure bounded storage on the communication arcs. For example, a valid schedule of an iteration for the SDF model in Figure 2.1 is AB.

2.2.1 Firing Vector

The first step in creating a schedule is determining the firing vector. The firing vector specifies the number of times each actor must fire during an iteration of the SDF model. For homogeneous graphs, the firing vector will indicate a single firing for each actor. For multi-rate graphs, actors may need to fire multiple times within an iteration.

The firing vector can be determined by solving the constraint equations associated with each arc in the model. The constraint equations ensure bounded storage on each arc in the graph. Specifically, these equations are created to ensure that each arc has bounded storage (i.e. no accumulation of tokens) and are deadlock free (i.e. an arc does not run out of tokens). These equations guarantee that the same number of tokens are produced and consumed on each arc during a single iteration.

To force this constraint, one equation is created for each arc. These equations take the form:

$$N_{source} * Q_{source} = N_{sink} * Q_{sink} \tag{2.1}$$

where N signifies the number of tokens consumed or produced for each firing, and Q is the number of firings of the given actor in an iteration. N is specified as part of the model and Q must be determined. The vector of all Q values forms the firing vector.

This process can be demonstrated by considering the constraint equations for the SDF graph from Figure 2.5. The constraint equations for the three edges are given below:

$$\mathrm{A} \rightarrow \mathrm{B} \qquad 2 * Q_A = 3 * Q_B \tag{2.2}$$

$$\mathrm{B} \rightarrow \mathrm{C} \qquad 1 * Q_B = 1 * Q_C \tag{2.3}$$

$$\mathrm{B} \rightarrow \mathrm{D} \qquad 1 * Q_B = 1 * Q_D \tag{2.4}$$

where Q_A, Q_B, Q_C, and Q_D represent the number of times the corresponding actor will fire in an iteration. While there many solutions to these equations, the smallest values for Q_A, Q_B, Q_C, and Q_D that satisfy all three equations are $Q = [3, 2, 2, 2]$. This indicates that actor A must fire three times during a single iteration and actors B, C, and D must fire two times during the iteration. These numbers represent the firing vector for this model.

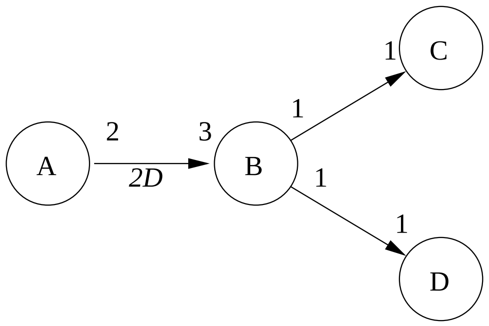

Figure 2.5: Sample dataflow model.

2.2.2 Iteration Schedule

The next step in the scheduling process is determining the ordering of actor firings. As stated earlier, this schedule must avoid deadlock by ordering the firings in such a way that all actors will have tokens on their input ports when they are scheduled to fire. Such an ordering will be impossible to find, for example, if a feedback loop exists which does not contain sufficient delays. An efficient way to determine the schedule is given in [10].

There are often many possible schedules for a given SDF graph. Different schedules can be optimized for different requirements, such as faster execution time or reduced memory usage. For example, consider a sequential implementation (using a single processor) of the SDF model in Figure 2.5. In the previous section, the firing vector $ABCD = [3, 2, 2, 2]$ was determined for this graph. Using this firing vector, the following schedules are all valid:

1. AAABBCCDD
2. AAABCDBCD
3. AABABCCDD
4. AABABCDCD
5. AABCDABCD

Other valid schedules are possible.

While each of these schedules are valid, some schedules may be more desirable than others due to target implementation constraints. A common implementation constraint is the minimization of storage for communication arcs. For example, schedule (1) requires the storage of six tokens on the arc $A \rightarrow B$ while schedule (6) only requires the storage of four tokens for this arc. Other implementation constraints may also affect the desirability of a schedule.

2.3 Implementing SDF

Thus far, SDF as an abstract dataflow specification has been discussed. However, in order to be useful, an SDF model must be implemented. For implementation, SDF algorithms can be targeted either to software or hardware. Software synthesis involves converting the SDF model into code that can be run on a sequential processor, such as a general microprocessor or a digital signal processor. Hardware implementation can be performed on programmable logic such as a Field Programmable Gate Array (FPGA), or with fixed logic such as an ASIC.

A dataflow graph is synchronous if all the ports on all nodes have token production and consumption rates that are known *a priori*. That is, the rates are not affected by the data, but are specified as part of the algorithm definition. Because they are known statically, it is much easier to synthesize software or hardware from SDF graphs than from general dataflow graphs. It allows compile-time (or synthesis-time) evaluation of execution order and memory usage.

2.3.1 Software Implementation

A software implementation of SDF models means that some kind of program code is executed on a processor. A software implementation of an SDF model is convenient since in general it is easier to design arbitrary actors. There are two main categories of SDF models that execute in software: those that execute on a single processor, and those that use multi-processor systems.

Ptolemy is one example of a software implementation of the SDF semantics. It is a Java program that executes a SDF model using a single program thread. Ptolemy will be described in Chapter 2.4. Other software implementations for SDF graphs are described in [11, 12, 13].

Software implementation of SDF systems on multiple processors is relatively easy as natural parallelism is exposed by the SDF algorithm. In this case, the schedule must, in addition to providing a actor firing order, provide the processor on which each firing occurs. Determining an efficient (but not necessarily optimal) parallel schedule is discussed in [10]. It is not necessary that multiple firings of the same actor within an iteration occur on the same processor, however improving locality could improve execution time.

Using a parallel processor to execute SDF can result in significant speedup as actors can be executed in parallel. However, in this case one must consider communication delays and synchronizing mechanisms to prevent a node on one processor from firing before it has received its input data from a node executing on a separate processor.

2.3.2 Hardware Implementation

Hardware can also be targeted in the implementation of SDF graphs. A hardware implementation can be much faster than software, but in general hardware is a more difficult synthesis target.

A useful property of SDF graphs is that scheduling can be done statically, at compile-time. This greatly simplifies the control mechanism for hardware implementations of SDF graphs. Control is needed so that each node only fires when it has the input tokens it needs. Methods of controlling hardware synthesized from SDF graphs are described in [14] and [15].

In synthesizing SDF graphs to hardware, the hardware description for each node is needed. Williamson[14] describes a hardware synthesis technique for SDF graphs which requires that SDF actors already have hardware implementations. Design in this case consists of specifying which library elements are to be used and connecting them appropriately. Although this is convenient from a synthesis standpoint, it limits the designer to only those blocks for which the implementations exist.

Another option is to let the designer also specify the behavior for nodes. The designer can then make the nodes fit the exact needs of the algorithm. A hardware description of each node is still needed for synthesis. An important part of this work is extracting a suitable hardware description from a node whose behavior is specified in software.

This work combines the benefits of software and hardware implementations. Software implementation has the ability to design arbitrary actors, while hardware implementations can execute much faster. This research will start with software descriptions of actor behavior. These descriptions will be synthesized to a form (called abstract circuit graphs) that can be easily converted to a hardware implementation.

2.4 Ptolemy

There are a number of tools available for specifying and modeling DSP systems with dataflow semantics. One such tool is the Ptolemy heterogeneous modeling and design system[16]. Other tools include GRAPE-II[17, 18], SPW from Cadence Design Systems and Cossap from Synopsys. The Ptolemy modeling framework is the tool that is used to express the SDF algorithms that will be synthesized by this work. This section will introduce the Ptolemy framework, and describe why it is used for this synthesis work.

The Ptolemy Project, led by the University of California at Berkeley, studies modeling techniques for concurrent heterogeneous systems[16]. The goal of this research effort is to facilitate the design of heterogenous embedded systems that mix analog, digital, and even mechanical components. The Ptolemy group at Berkeley has created a number of tools and techniques for modeling heterogeneous systems[19].

The principle behind this framework is the use of well-defined *models of computation* that define the interactions between components (also called actors) within a model. Models of computation, implemented in software packages called “domains”, define the rules of communication, synchronization, and execution of concurrently operating actors. The Ptolemy-II modeling environment currently includes the following models of computation: Synchronous Dataflow (SDF), Discrete Event (DE), Communicating Sequential Processes (CSP), Continuous Time (CT), Finite-State Machines (FSM), and others.

The original modeling tool, called Ptolemy Classic, was developed in C++ and provided several models of computation. More recently, the Ptolemy group has created a Java-based modeling framework called Ptolemy II [20]. This tool contains a set of Java packages and design aids that supports the modeling and design of heterogeneous, concurrent systems.

2.4.1 SDF Domain

Ptolemy-II provides a dedicated SDF "domain" that implements the SDF model of computation. This domain provides a library of useful pre-defined SDF actors, communication primitives, and an SDF actor scheduler. This scheduler statically schedules the execution of SDF actors to properly simulate the behavior of a concurrent SDF model on a sequential processor.

An example of an SDF model in Ptolemy is shown in Figure 2.6. This model implements a cascaded integrator-comb or CIC filter. CIC filters are frequently used for narrow band filtering when large sample rate changes are required[21]. This model contains two instances of a discrete integrator, a downsample rate change, and two instances of a comb filter. SDF semantics are specified in this model with the inclusion of the SDF director.

2.4.2 Actors

In Ptolemy, the nodes in a model are called *actors*. There are two kinds of actors: composite and atomic. Hierarchy is implemented with composite actors, as they can contain other actors. A composite actor may also contain a different director, which would define the model of computation for the actor nodes contained by the

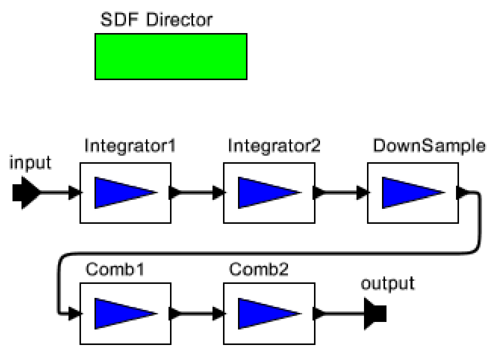

Figure 2.6: Ptolemy II SDF model.

composite actor. The computation in a model takes place in atomic actors. The actors synthesized in this thesis are atomic actors.

Atomic actors within Ptolemy II are created by writing a Java class that conforms to specifications outlined in the Ptolemy II design guide. Actor classes will create ports for the actor, specify internal parameters, and define the behavior of the actor using several action methods. The most important action methods for the SDF domain include the `prefire`, `fire`, and `postfire` methods. The initial state of the actor is determined by three initialization methods: the constructor, `preinitialize`, and `initialize`. A simplified Java template for an SDF actor is shown below in Listing 2.1.

2.4.3 Actor Operation in Ptolemy

In order to create a dataflow graph from a Ptolemy actor, it is important to understand how these actors work. There are five main methods of interest, besides the constructor. When a model is initialized, it initializes each of the actors in it. The initialization procedure is important because it determines the initial state of the actor.

```
public class SimpleSDFActor ... {
  ...
  public SimpleSDFActor() ... //The constructor
  public void preinitialize() ...
  public void initialize() ...
  ...
  public void prefire() ...
  public void fire() ...
  public void postfire() ...
  ...
}
```

Listing 2.1: Ptolemy actor action methods.

The constructor runs when the actor is instanced. The constructor typically sets up the ports and parameters for the actor. The next method called in the actor is the `preinitialize` method, followed by `initialize`. Ports can be dynamically typed in Ptolemy, and the main difference between `preinitialize` and `initialize` is that Ptolemy has completed its typing procedure between the two calls. This distinction is not important for synthesis, only the order in which they are called.

The constructor, `preinitialize`, and `initialize` methods are called once at the beginning of the simulation to setup the actors. An actual *firing* of the actor involves three more methods: `prefire`, `fire`, and `postfire`.

The general semantics of Ptolemy are that `prefire` will be called until it returns true, `fire` could be called an indeterminate number of times, and `postfire` will be called exactly once per actor firing. Also, this actor will not be fired again if `postfire` returns a false value.

The behavior under the SDFDirector, however, is assumed to be more simple for purposes of synthesis. Under SDF semantics, the only requirement for a node to fire is to have input tokens on all its incoming arcs. Ptolemy guarantees this will be the case when it creates the firing schedule. The `prefire` method is assumed to be called exactly once per iteration. The `fire` method is also executed once. Since

SDF algorithms are designed to handle a theoretically endless data stream, it will be assumed that `postfire` will always return true, meaning that another iteration can take place. These assumptions simplify the hardware synthesis.

2.4.4 Code Generation

The flexibility that Ptolemy allows in having many different models of computation comes at the cost of performance. For example, to get from one actor to another, data must be encapsulated in a Token object, sent using a Port object, which passes it to the Receiver for this particular model of computation. The receiving Port object queries the Receiver, and then the Port finally passes the Token to the Actor for this node. All these levels are necessary for a fully flexible system, but it does slow things down.

A solution is to generate code specifically geared to execution of one particular model that does away with much of the overhead mentioned above. Such streamlined code could execute much faster than the normal Ptolemy framework, but still have the same semantics and results as the model was designed. Fast, efficient execution is especially important for embedded systems, which is a primary focus for Ptolemy.

Code generation produces a self-contained executable program, so it is suitable as a final implementation of a model being simulated in Ptolemy. The program can then be compiled to run on an embedded system. Copernicus is a code generation tool for the Ptolemy system[13]. Copernicus utilizes actor specialization along with traditional compiler optimizations to do code generation. Actor specialization increases execution speed by specializing an actor to run in a particular instance of a model by removing a lot of support that allows full general execution (such as supporting multiple data types, multiple models of computation, etc.). The hardware synthesis discussed here will take as input the bytecode from the Java code generation technique and synthesize it into a graph representation that can be easily converted to hardware.

2.5 Conclusion

In this chapter the synchronous dataflow computation model has been presented. SDF is a useful means of expressing algorithms that are data-driven and parallel. A property of SDF that makes it beneficial for synthesis is that token rates are specified as part of the model. SDF models can be implemented in software or hardware. Software is easier to design and run, but a hardware implementation has the potential to execute much faster. One example of a software implementation of the SDF model of computation is included in Ptolemy II. Ptolemy is a modeling system that supports many different models of computation, among them SDF. The SDF models to be synthesized by the technique presented in this thesis are specified in Ptolemy, and the actors to be synthesized conform to the Ptolemy guidelines. The behavior for these actors will be extracted into dataflow graphs, which will then be combined according to the topology of the SDF model.

Chapter 3

Generating Dataflow for SDF Actors

An essential part of this synthesis process is extracting the behavior of the actors defined in software and generating a corresponding SDF representation. Unlike previous efforts to generate hardware from SDF, this approach will synthesize hardware from actors defined in software code (as opposed to having a predefined library of actors). This chapter will detail the process of extracting the dataflow behavior from actors.

Ptolemy SDF actors are described in Java and contain standard control flow primitives found in most sequential programming languages. In order to perform synthesis, the Java actors must conform to the limitations discussed in Section 3.7. The synthesis process described in this chapter will remove all control flow primitives (i.e. branching) from an SDF actor. This process will result in a purely dataflow description of the actor behavior. All subsequent dependencies are data dependencies which exposes the maximum available parallelism within the actor code.

This chapter will describe the process used to generate the dataflow representation of the actor from its original control-dominated description. The first step of this process is disassembling the compiled bytecode of the SDF actor. A bytecode analysis tool, Soot, will be used to transform a Java method into a Control Flow Graph (CFG). All methods that are executed during an actor's firing will be combined into a single CFG. To begin the processes of removing control flow dependencies from the CFG, intervals are identified within the CFG. This interval analysis identifies the scope of control flow primitives to resolve mutually exclusive variable definitions. After interval analysis, the dataflow of each basic block is generated and combined to produce the dataflow graph. Further analysis will identify internal actor state and identify the ports needed for actor communication. This chapter will conclude by discussing the limitations of this approach. The dataflow extraction process is summarized below:

1. Disassemble bytecodes of compiled actor class file
2. Create CFG of each action method and combine CFGs
3. Perform interval analysis on control flow graph
4. Extract and merge dataflow of each basic block
5. Identify internal actor state
6. Identify ports (actor I/O)

The process of generating an SDF graph from an actor will be demonstrated with an example. The code segment shown in Listing 3.1 performs a simple three-tap FIR filter operation. In this example, all of the actor behavior is described in the `fire` method[1]. This method begins by reading the integer value of the token on its input port. Next, the FIR operation is performed by executing three multiply-accumulate operations. The result of this operation is clipped using constant `MIN` and `MAX` output limits. The final result is sent on the actor's output port. After completing this computation, the internal class fields `delay1` and `delay2` are updated with new values to store the persistent delay line state.

3.1 Java Disassembly

The first step in this process is to disassemble the bytecodes found in the Java `.class` file for a given SDF actor. The format of the Java `.class` file is widely available[22] and contains all the information necessary for extracting the dataflow

[1]Many of the necessary details are omitted for brevity.

```
public class SimpleFIR ... {
   ...
   public void fire() {
     IntToken in = (IntToken)input.get(0); // Get token

     IntToken mac = (IntToken)in.multiply(c0);
     mac = (IntToken)mac.add(delay1.multiply(c1));
     mac = (IntToken)mac.add(delay2.multiply(c2));

     if (mac.isGreaterThan(MAX).booleanValue())        // clip result
       mac = MAX;
     else if (mac.isLessThan(MIN).booleanValue())
       mac = MIN;

     output.send(0, mac);    // Send result

     delay2 = delay1;      // update memory
     delay1 = in;
   }
   ...
}
```

Listing 3.1: Simple Ptolemy FIR filter actor.

and control flow of the actor behavior. Analyzing precompiled Java binary files avoids the need to parse Java source code and create the necessary syntax tree and symbol table information.

Another project that attempts to synthesize hardware from Java bytecodes is the JHDL synthesis environment created by Carl Worth[6]. This project synthesizes hardware from programs written in Java according to the to the JHDL specification [8]. In the JHDL synthesis project, a custom bytecode analysis technique was created. This bytecode analysis tool decoded the bytecodes of binary Java class files and provided rudimentary dataflow analysis support.

In this project, Java disassembly step is accomplished using the Soot Java Optimizing Framework[23]. The Soot optimizing compiler is a framework that can analyze the Java bytecodes and manipulate these bytecodes using a rich API. Several of the steps described in this paper rely on program code analysis and manipulation techniques provided by the Soot compiler framework.

Soot reads the `.class` file and represents it with Java objects. Each bytecode instruction is represented by a subclass of the Soot `Unit` class. The units are chained together in the order of the bytecodes in the class file. Soot provides methods to analyze and manipulate this unit chain.

To facilitate bytecode optimizations, Soot supports several different bytecode intermediate representations. The representation used here is called Jimple, which is three address code similar to RISC assembly instructions. Each Jimple statement has a destination operand, up to two source operands, and an operation. These operations are Java primitive operations, such as add, subtract, logical shift, etc. These primitive operations will be the nodes of the final circuit representation.

Jimple is used because it is easily analyzable for dataflow extraction. Bytecode instructions use a stack to locate their operands. Determining where the operands for a particular instruction were written can be cumbersome, since an arbitrary number of instructions could separate the writing of a value to the stack and it being read

```
public int compute(int a, int b, int c) {
   int d = (a + b) * 3;
   return d - c;
}
```

Listing 3.2: Sample Java source code.

again. The stack is also not typed, although it is possible to statically determine the types of the values on the stack. Soot performs stack analysis when converting to Jimple. Typed, local variables are created to represent stack values. The bytecode operations are then converted to their three address equivalents. Listing 3.2 demonstrates a Java method with a simple arithmetic expression. Listing 3.3 demonstrates the corresponding bytecodes and their Jimple representation.

```
   0 iload_1                   r0 := @this: temp;
   1 iload_2                   i0 := @parameter0: int;
   2 iadd                      i1 := @parameter1: int;
   3 iconst_3                  i2 := @parameter2: int;
   4 imul                      $i4 = i0 + i1;
   5 istore 4                  i3 = $i4 * 3;
   7 iload 4                   $i5 = i3 - i2;
   9 iload_3                   return $i5;
  10 isub
  11 ireturn
```

Listing 3.3: Sample Java bytecode.

There are several interesting differences between the bytecode representation and the Jimple representation. The behavior of the method is represented as stack operations in the bytecode representation. Each operation implicitly pops its arguments from the stack and pushes the result back onto the stack. The Jimple code uses a standard three address format for representing the behavior. These three addresses represent two source arguments and a destination location. Additional local variables are created to represent stack locations. It is easier to create a dataflow graph from Jimple code since the arguments of an operation are more explicity identified.

3.2 Control Flow Analysis

After disassembling the bytecode of the actor, a control flow graph is created for each actor action method. A control flow graph is a directed graph representation of a method where the nodes correspond to basic blocks and the arcs represent possible paths of execution. Branching statements such as `if` or `switch` identify mutually exclusive execution paths within the program execution. These branching statements are represented in the CFG by multiple arcs leaving a control flow node. Multiple arcs leaving a control flow node represent mutually exclusive paths of operation and identify the various targets of the branching statement. The actual execution path is determined at runtime by the branching condition.

The control flow graph for a given actor action method is created by the control flow analysis routine `BriefBlockGraph` provided by Soot. After disassembling the bytecodes of the method, the CFG is created in Soot by identifying basic blocks and determining the control paths from branching statements.

The control flow graph for the `fire` method of the SimpleFIR example of Listing 3.1 is shown in Figure 3.1. This method contains five basic blocks and two control flow branches. The function of each bytecode within the basic blocks is shown inside of the basic block nodes. The branching condition (true vs. false) is identified on mutually exclusive control flow paths.

The runtime behavior of a Ptolemy actor is contained in only a few methods. These methods will be referred to as the "action methods." These action methods are `prefire`, `fire`, and `postfire`. A control flow graph will be created for each method and the individual graphs will be merged into a single CFG representing all the firing behavior of the actor.

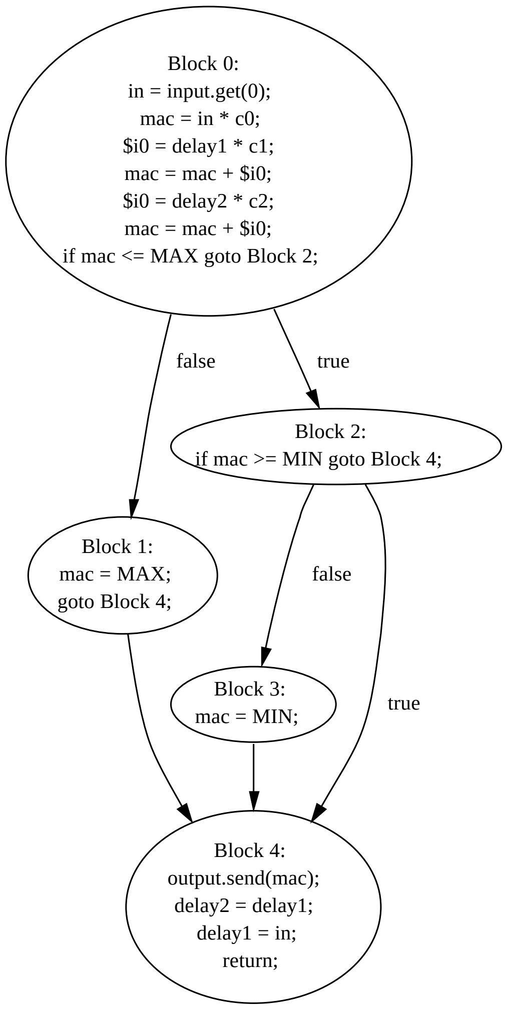

Figure 3.1: Control flow graph for SimpleFIR.

The CFGs of the action methods are merged by placing all CFGs in a single graph, and inserting a precedence arc between the sink node of the preceding CFG to the source node of the following CFG. Each action method is executed once when firing an actor as dictated by the semantics of the Ptolemy SDF director. As described earlier, the `prefire` method is executed first, followed by the `fire` method, and finally completed with execution of the `postfire` method. To implement these semantics in a single CFG, the sink node of the `prefire` method is connected to the source node of the `fire` method, and the sink node of the `fire` method is connected to the source node of the `postfire` method. The final result of this control flow analysis is a single control flow representation of the actor firing behavior.

3.3 Interval Analysis

One of the goals of this synthesis process is to extract as much parallism as possible from the actor action methods. In order to achieve this goal, *all* mutually exclusive control flow paths will be executed in parallel. The results from mutually exclusive control flow paths will be computed in parallel along with the branching condition expression. Once the branching condition has been resolved, the results from the correct control flow branch will be propagated. This form of predicated execution will be respresented as a single dataflow graph with multiplexer nodes added to choose the appropriate results.

An important step in the process of creating this monolithic dataflow graph from the control flow graph of a method is identifying the *intervals* of the control flow graph. Intervals are defined in [24] as a header node plus all the nodes that the node dominates. A node A dominates a node B if every incoming path to node B goes through node A. In this thesis, an additional constraint is placed on intervals such that if all paths exiting the header node of an interval merge into a single node, the interval ends. Thus an interval within the control flow graph is defined as a set of connected control flow nodes with one entry node and one exit node.

An interval can be as simple as a single control flow node with one entry point (i.e. entry into the node) and one exit point (i.e. exit from the node). More interesting intervals contain multiple nodes organized around a single entry point and a single exit point. Multi-node intervals may contain control flow constructs such as a branch. Such an interval contains a branch condition and all nodes that could be executed as a result of the branch. This interval must also include a node that merges the mutually exclusive control paths leaving the branching node.

The interval is similar to the hyperblock[25] and the superblock[26]. Both the hyperblock and superblock represent a collection of related control flow nodes with a single entry point. However, the hyperblock and superblock differ from the interval by allowing multiple exit points. In a superblock all nodes must be from the same control flow path while a hyperblock may contain nodes from mutually exclusive control flow paths.

The SeaCucumber project[7] uses hyperblocks to perform synthesis. The bytecodes in a hyperblock are converted into a form called Predicated Static Single Assignment (PSSA). PSSA[27] is a technique first applied to compiler optimizations for VLIW computer architectures. The goal of PSSA is similar to the approach detailed here, in that false data dependencies are removed, exposing maximum parallelism. The SC project synthesizes the PSSA-transformed hyperblocks into hardware.

PSSA creates and renames variables such that all variables have a single assignment. Then each statement is predicated with the conditions that must be true for the results of that statement to be committed. In this way, instructions from multiple control flow paths may execute simultaneously while their guarding predicates are also determined. At the end, the instructions that should have executed have their results committed.

PSSA is similar to the interval analysis technique here, but their are important differences as well. These techniques share similar goals of removing false data dependencies. The path predicates in PSSA are analagous to the labels presented here. Both contain the information necessary to know if the instructions in a certain control path should execute. However, PSSA uses hyperblocks as its basic unit, while this approach uses intervals. Variable resolution in this approach happens at the end of an interval(where multiple control flow paths join together), while in the PSSA approach variable values aren't resolved until the variable is assigned. Thus, if a statement requires a value defined in multiple control flow paths, that statement will be replicated multiple times in PSSA. Each copied statement will operate on a value from a different control branch. By contrast, this approach resolves variables when they are read. The PSSA approach may potentially find more parallelism due to resolving later, but at the expense of additional operations.

Unlike the hyperblock and superblock, interval boundaries are not always found naturally within a control flow graph. A branching condition that creates a fork in the control flow graph does not always have a corresponding join node in the control flow graph. In many cases, multiple branching conditions are resolved at the same node, which prevents the identification of a single interval for each branching condition. The interval analysis process must add nodes to the CFG to create explicit intervals for each branch condition.

Intervals are used to resolve the multiple definitions of variables in mutually exclusive control flow path. At the conclusion of an interval, the variable definitions made within the different paths internal to the interval will have to be resolved. Because there is only a single exit point from the interval, the actual path of execution through mutually exclusive control paths is resolved when leaving the interval. Once the actual control path is known, the variables defined in the given control path are propagated or passed on to subsequent operations in the dataflow graph. Any subsequant operation that depends on a variable defined in this interval can obtain the correct value of the variable. Variable resolving is performed at the exit point of the interval when the mutually exclusive paths join together. An explicit node to represent this joining is inserted in the CFG.

Figure 3.2 shows a simplified version of the CFG for the SimpleFIR actor. This figure illustrates why it is necessary to identify control intervals. There are three execution paths entering Block 4. In other words, there are three different ways the program can reach this point. Each of the three paths may assign a different value to a given variable. Blocks 1, 2, or 3 could have written a value to a variable that is used in Block 4. Before performing any computation in Block 4, the actual execution path needs to be determined to resolve mutually exclusive variable definitions. Interval analysis is used to determine which value should be used. To illustrate this problem more concretely, consider the code in Listing 3.4.

```
1:  if (a > b){
2:    a = 1;
3:  } else {
4:    a = 2;
5:  }
6:  c = 2 * a;
```

Listing 3.4: Variable resolution example.

The assignment to the variable `c` on line 6 depends on the value of `a`. The value of `a` is set in two mutually exclusive control paths (i.e. lines 2 and 4). This control path is determined by the `if` conditional in line 1. In a serial execution, the condition will be determined first, and only one assignment to `a` will be made. In this hardware implementation, both paths are executed simultaneously producing *two* values for the variable `a`. Before proceeding to line 6, the actual value of `a` must be resolved. The interval containing the two mutually exclusive control paths will end before the assignment to `c` in line 6. The value of variable `a` is resolved at the end of this interval. Interval analysis is used to find the interval exit points in all mutually exclusive control paths.

To identify intervals, the entry and exit nodes must be found. The entry node for an interval is clearly identifiable because they are the nodes that end in an `if` condition. Interval exit point is where the mutually exclusive control paths created by the branch join together. After this interval exit point, the branch condition no longer effects the flow of control (i.e. the branch condition is out of scope).

In order to perform this analysis, interval exit points must be determined. However, interval exit points are not explicit in the original CFG. Specifically, multiple

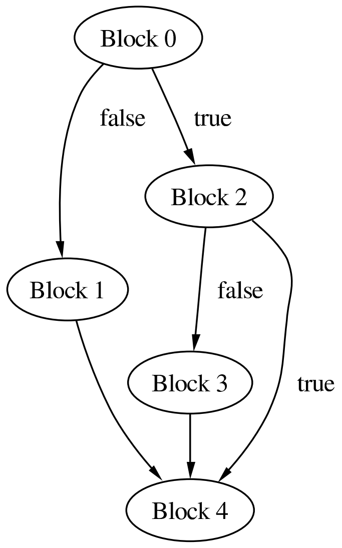

Figure 3.2: Control flow graph for SimpleFIR actor.

control flow arcs may enter a single CFG node (see Block 4 in Figure 3.1). Variable definitions in mutually exclusive control paths must be resolved *before* entering a CFG node. The interval analysis will identify interval exit points and insert JOIN nodes within the CFG to explicitly mark the end of an interval. These JOIN nodes will determine the variable resolution.

The JOIN node will contain the condition necessary to discern which path of execution should have been taken. The JOIN can be thought of as similar to a multiplexer. It has three ports: a true port, a false port, and a condition port. It will pass through the values only from the path that the condition port indicates should be taken. This allows both paths (along with the condition itself) to be executed in parallel, and only the correct results will be passed through.

A fork/join paradigm is used, where the fork is the `if` statement and splits into two possible paths. The JOIN node is explicitly inserted by this algorithm when the `if` block completes. The challenge lies in determining where the `if` block terminates so a JOIN node can be added.

The intervals are discovered through a process developed for this thesis called *labeling.* A label is attached to each execution path that represents all the conditions that affect the execution of a given path. When multiple paths enter a node, the labels are examined to see if they match. Two paths have matching labels when the latest condition on each path is the same. This means that the two branches of a `if` statement are joining back together, so the condition is no longer in scope. It also means that the interval begun by the `if` condition is ending. The two paths are connected by a JOIN node, and the matched condition is removed from the label. For a detailed discussion of this labeling algorithm, see Appendix A.

To demonstrate this interval analysis, consider the CFG shown in Figure 3.1. Figure 3.3 demonstrates the arc labeling of the SimpleFIR CFG. These labels are used to find interval exit points. For example, Block 4 has more than one incoming path and indicates the exit point of at least one interval. The labels on the arcs reaching Block 4 are examined and matched to determine the appropriate interval exit. The two arcs entering Block 4 on the right are identified as a match. Since these two arcs represent the mutually exclusive control paths created in Block 2, this matching process has identified the end of the interval that started in Block 2. A JOIN node is added to mark this interval exit point. The two matched paths become inputs to the JOIN node, and the output of the JOIN contains a new label that is common to both paths. An arc is also added between Block 2 and the JOIN node indicating the data dependence on the condition for this JOIN. The modified graph with the new JOIN node is shown in Figure 3.4.

After identifiying this first interval, there still remain unresolved interval exit points (i.e. two paths entering Block 4). The two paths entering Block 4 are matched since their labels were both created by the condition expression in Block 0. These two paths are resolved with another JOIN node. This JOIN node marks the exit point of the interval initiated in Block 0. Once this node has been added, there are *no* CFG

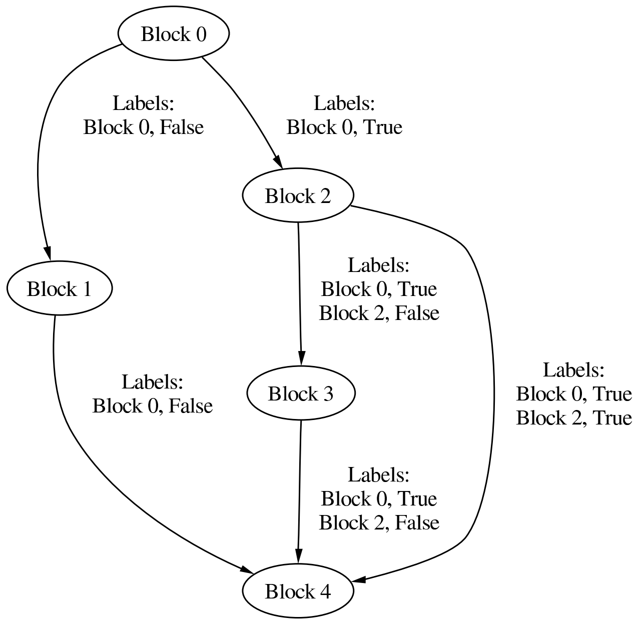
Block 0
Labels: Block 0, False
Labels: Block 0, True
Block 2
Block 1
Labels: Block 0, True Block 2, False
Labels: Block 0, False
Block 3
Labels: Block 0, True Block 2, True
Labels: Block 0, True Block 2, False
Block 4

Figure 3.3: Labels when Block 4 is being processed.

nodes with more than one entry arc. This ends the inverval analysis process. Figure 3.5 shows the graph when interval analysis has completed.

3.4 Dataflow Analysis

At this point in the synthesis process we have created a CFG representing all of the actor's control flow during firing. In addition, intervals were identified within the CFG to mark the location in which variable resolution must take place. The desired result of this synthesis step is to generate a dataflow graph for the actor that contains no control flow and whose nodes represent primitive operations. This section will describe how such a dataflow graph is generated.

There are two specific steps in this dataflow analysis process. First, a dataflow graph is generated for each node in the original CFG. These nodes represent a basic block. Second, the dataflow graphs for each basic block must be merged according to

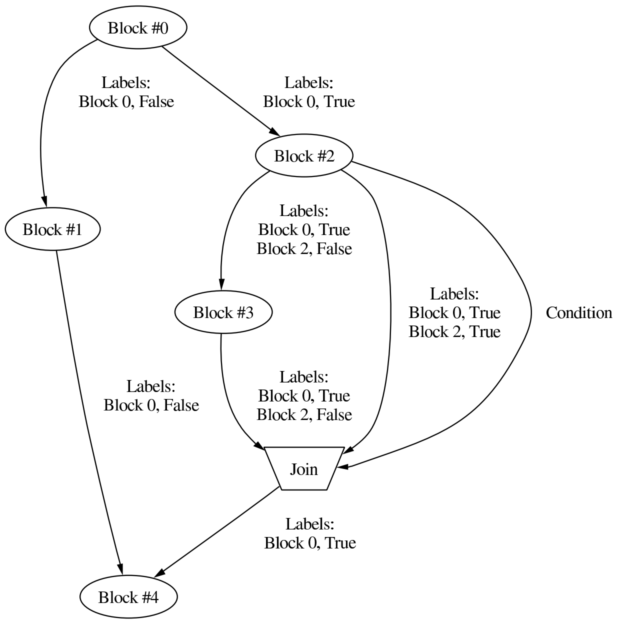

Figure 3.4: First JOIN added during interval analysis.

the topology of the CFG created in the previous section. Each of these steps will be discussed below.

3.4.1 Extract Dataflow from Basic Blocks

The first step of the dataflow analysis is to generate a dataflow graph for each basic block. Nodes within the basic block dataflow graph represent primitive Java operations. By definition, a basic block is a block of code (Jimple statements in this case) with no jumps in or out of the block. There is only one entry point to a basic block (the first statement) and one and only one exit point (the last statement). Since basic blocks contain no control flow, it is fairly straightforward to generate a dataflow graph of the operations performed within the basic block.

A dataflow graph is created by evaluating the Jimple statements of the basic block in sequential order. A new node is created within the dataflow graph for

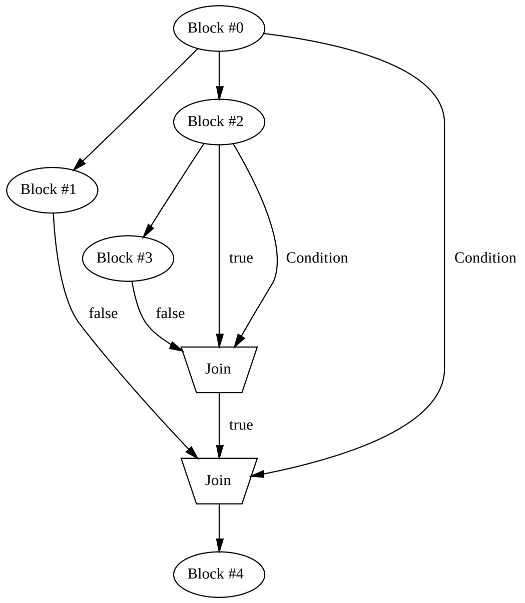

Figure 3.5: CFG after interval analysis.

each statement. The operation of the new dataflow node corresponds to the operation performed by the corresponding bytecode. Nodes are also created for variable assignments and class field references.

The arcs of the dataflow graph are created by evaluating the data dependencies between bytecode operations. These data dependencies are determined by identifying the source operands for each byte code and adding an edge between each source operand and the corresponding operation. For example, consider the Jimple code from Block 0 from the SimpleFIR actor in Listing 3.5.

These statements are processed in order. The first statement assigns the field `input` to the local `r4`. A node is created for these two values, and a directed arc is created from the node for `input` to the node for `r4` showing the assignment. Line 3 is then processed, which is a method call to the `get` method on the port `input`. Nodes are created for the method call as well as the parameters (the channel number for the

```
r4 = r0.<ptolemy.copernicus.jhdl.cg.FIR.CGFIR:
        ptolemy.actor.TypedIOPort input>;
r5 = virtualinvoke r4.<ptolemy.actor.IOPort:
        ptolemy.data.Token get(int)>(0);
r1 = (ptolemy.data.IntToken) r5;
r6 = <ptolemy.copernicus.jhdl.cg.FIR.CGFIR:
        ptolemy.data.IntToken c0>;
r7 = virtualinvoke r1.<ptolemy.data.ScalarToken:
        ptolemy.data.Token multiply(ptolemy.data.Token)>(r6);
```

Listing 3.5: Jimple code sample for Block 0 of the SimpleFIR actor.

`get` call, which is a constant zero in this case). A directed arc is added between the method call and each parameter node. Further, an arc is added between the class object (`r4` in this case) and the method node. A node is created for the return value `r5`, and an arc created from the method call to the local that stores the return value. This process continues for the remaining statements in the basic block. Figure 3.6 shows the extracted dataflow for this section of code.

In many Ptolemy actors, operations are performed on `Token` data objects rather that primive java types. The operations are performed on Token objects through methods of the `Token` class. In this synthesis process, the operation methods of the `Token` class and its subclasses will be replaced by a node representing the primitive operation.

Figure 3.7 demonstrates the complete dataflow graph for `Block 0` of the SimpleFIR actor. Many of the intermediate nodes that represent locals and stack variables have been removed to increase clarity. This dataflow graph contains three multiplies, two additions, and a comparison operation. Extracting the dataflow from this basic block exposes parallelism for the multiplication operations. Two of the multiplication operations have operands that do not depend on previous operations. With dedicated execution units, these multiplications could be executed at the same time as the first multiply, thus saving execution time.

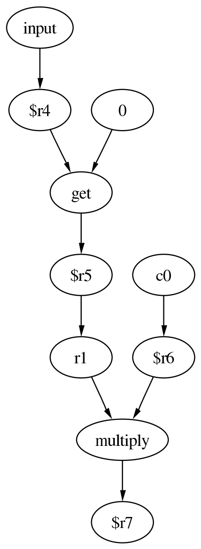

Figure 3.6: Example extracted dataflow.

3.4.2 Merge Dataflow Graphs

The previous dataflow analysis step will create a distinct dataflow graph for each basic block in the CFG. In order to create a single dataflow graph, these basic block dataflow graphs will be merged according to the topology of the CFG. This final dataflow graph represents the complete behavior found in the given actor source code. This section will describe the process of merging dataflow graphs.

Merging builds the dataflow graph bottom-up by starting with the final output values of the actor and identifiying all nodes necessary for computing these output values. This bottom-up process traverses the CFG and the basic block DFGs to find all necessary nodes in the graph. The algorithm used to create this graph is shown in Listing 3.6.

The first task (line 1 of Listing 3.6) is to identify the output values. The output values are those values sent through an output port. In the Java code, an output value is identified through `send` method calls on port objects. One parameter

```
Q = list of output values
S = CFG sink node
for each q in Q{
  inquire(S,q);
}

inquire(DataflowCFG node, value v) {
  last_def = last definition of v in this basic block
  CompositeDFG.insert(last_def)
  new node_search_list
  new sources_list
  node_search_list.add(last_def);

  for each val in node_search_list {
    if (val is a source in the node.DFG){
      sources_list.add(val);
    } else {
      preds = predecessor(s) of val
      node_search_list.add(preds);
      CompositeDFG.insert(preds);
    }
  }
  if (node.previousCFGNode == null)
    return last_def;
  for each src in sources_list {
    inquire(node.previousCFGNode, src);
  }
  return last_def
}

inquire (JoinNode node, value v) {
  true_val = true_port_node.inquire(v);
  false_val = false_port_node.inquire(v);
  condition_val = condition_port_node.inquire(JOIN node's condition);
  mux = new mux_node(true_val, false_val, condition_val)
  CompositeDFG.insert(mux);
  return mux;
}
```

Listing 3.6: Dataflow merging algorithm.

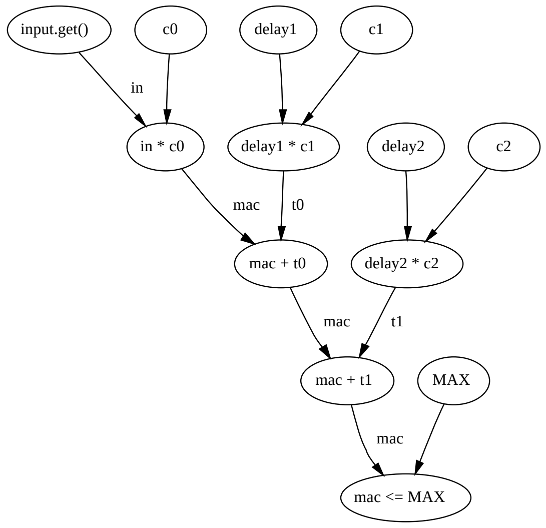

Figure 3.7: Dataflow graph of Block 0.

of the `send` method is the value that is to be sent through the output port. Dataflow merging will start with these method calls.

The merging process will begin with the sink node of the actor CFG. The dataflow is performed bottom-up, and as the sink is the bottom of the graph, it is a logical place to start merging. The sink node is inquired for each of the output values.

There are two types of nodes in the CFG following interval analysis. Basic block nodes are those nodes that were a part of the original CFG and represent basic blocks of the Java bytecode. All of the operations for the actor are defined in these basic block nodes. Added during interval analysis are the JOIN nodes which mark the end of intervals. In these nodes the resolution of assignments to variables in mutually exclusive control paths must be resolved. Each block behaves differently when inquired for a data value. The overall algorithm for inquiring both basic block nodes and JOIN nodes is shown in Listing 3.6.

A process of *inquiring* CFG nodes forms the core of the merging algorithm. Inquiry is a demand-driven[28] process. Previous demand-driven analysis work has focused on software optimization[29, 30]. In this work, demand-driven analysis will be applied to hardware synthesis. Only those operations necessary to determine the outputs will have their dataflow extracted. A node is inquired to determine what operations that node has which contribute to the output.

When a CFG node is inquired, a value is passed to it for which the dataflow operations used to obtain that value are desired. The CFG node will determine the appropriate action to take based on the type of CFG node it is. Basic block nodes will examine the basic block dataflow graph associated with their node and extract those operations and intermediate values that are used to define the passed in value. If the operation to define the passed in value or any intermediate value is not found in the basic block node, the node then inquires its predecessor node for the value. The CFG source node will not have any predecessors, and so inquiring will terminate at the source. When a JOIN node is inquired, it inquires both the nodes connectecd to its true and false input, and then inserts a multiplexer in the composite DFG to choose between the value computed in the true and false paths. The condition port node is inquired to determine the value of the condition used to choose the correct path.

The process for basic block node (also called dataflow node) inquiring is shown starting in line 7. A dataflow node will be passed in a value for which the dataflow should be found. The node will examine its corresponding dataflow graph to see if it defines (writes to) the value. If so, the last assignment to that value in this basic block is found. The node representing that assignment is added to the composite DFG. The last definition starts the list of nodes to be searched for in the current basic block. Then for each value in this list, the operation that determines the value and source operands of that operation are added to the composite DFG. The source operands are added to the list of nodes to search for. These source operands are the intermediate values for defining the passed in value. If a value in the search list is a source of the basic block DFG (that is, it has no predecessor nodes), then the operation that defines this value is located in a different basic block. These nodes are added to a list of source nodes that will be inquired of the predecessor node. If no predecessor CFG node exists, then inquiring terminates. Otherwise, the source nodes are inquired of the preceeding CFG node to find the operations that define the source nodes. Finally, this dataflow node returns the DFG node for the last definition of the passed in value. Figure 3.8 shows the result of the `mac` value being inquired at Block 0.

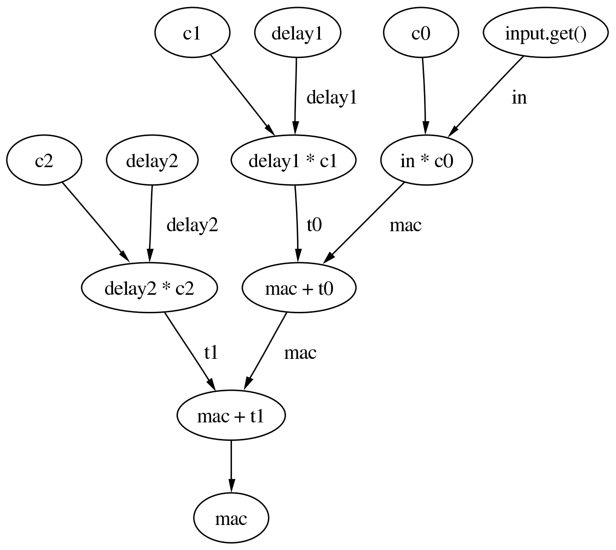

Figure 3.8: The `mac` value inquired at Block 0.

The inquiry process for a JOIN node is shown in Listing 3.6 starting in line 31. When a JOIN node receives a request for a data value, it makes inquiries for that value from the nodes attached to its true port and false port. The node attached to the condition port is also inquired, but that node is inquired for the condition value. The condition is a boolean value, and that value can be determined the same as a regular data value. It then joins the two returned DFG nodes with a multiplexer whose condition is the DFG node returned from the condition port. This new mux node is added to the DFG and returned.

Recall Figures 3.1 and 3.5, which show the SimpleFIR actor. In Block 4, the value `mac` is sent out through the output port. The value `mac` is inquired of beginning at the sink node (which also happens to be Block 4). Block 4 does not define `mac` (it only reads it), so it inquires of its predecessor node for `mac`. The predecessor is a JOIN, which always inquires of the two nodes that are connected to its data inputs. One of the JOIN's predecessors is Block 1. Block 1 assigns `MAX` to `mac`. `MAX` is a known constant, so no more inquiring is needed. The JOIN's other predecessor is another JOIN, which repeats the process. The JOIN nodes also inquire for their condition values so it knows how to choose between the data value inputs. The final dataflow graph for the `mac` value is shown in Figure 3.9.

3.5 Extract Internal Actor State

Many actors described within Ptolemy contain internal state. Actor outputs computed during the current iteration can depend on current inputs as well as inputs read in past iterations. The inputs or computations on inputs from past iterations can be stored as actor state. State is any value that persists from one firing of an actor to the next. In order to properly extract the behavior of Ptolemy actors, the state must be extracted.

In a Java actor, object fields can be used to store data between calls to the given action method. The SimpleFIR actor, for example, requires persistent state to implement a delay line for the input samples. The class field members `delay1` and `delay2` store the previous two input samples and are used to compute the FIR output. At the start of the `fire` method, the value of these class fields are read and used to compute the output. After the output value has been sent, these class field members are updated to reflect their new state.

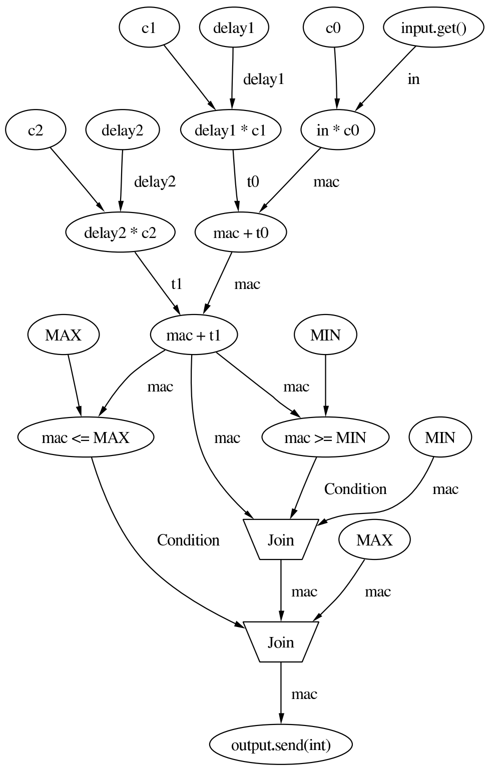

Figure 3.9: Dataflow for the `mac` variable.

Persistent state must be identified and extracted from the method. State storage is represented within the DFG by using *delay* nodes. A delay node acts much like a hardware register – the output of the arc is delayed one sample period. The state extraction process must identify persistent state within the actor code and insert the appropriate delay nodes in the final DFG.

Persistent state is identified within the actor by detecting object field variables that are read (i.e. a data source) *and* written in the same iteration. Reading a value of a field member corresponds to accessing the current state of the state variable. Writing a new value to the field member corresponds to updating the state variable with new state for the next iteration.

The DFG is analyzed to determine the input and output of the persistent state delay nodes. The input of the delay node (i.e. the value that is to be written into the persistent state, or next state) corresponds to the *last* value written to the object field in the firing. The output of the delay arc (i.e. the current state value) corresponds to the first use of the class field in the firing. Once these two nodes are identified, a delay node can be added to the SDF graph.

The current state values are those fields which are sources of the DFG after the output values have been extracted as outlined above. Notice in Figure 3.9 that `delay1` and `delay2` are sources of the DFG. To identify the next state, the last value written to the the fields can be found by extracting the dataflow for those fields, just as the output values were extracted. A delay node is then inserted with the next state as its input and current state as its output. Figure 3.10 shows a segment of the dataflow graph with the delay elements inserted.

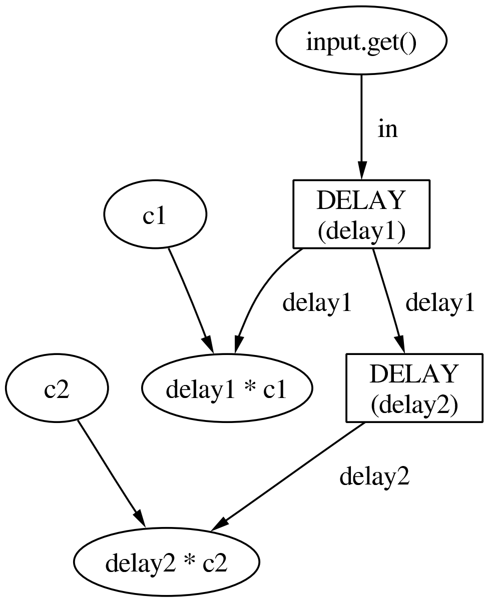

Figure 3.10: Portion of dataflow graph showing delay elements.

3.6 Identify Ports

Each Ptolemy actor contains ports that identify the actor inputs and outputs needed for a computation. During the firing of the actor, input ports *receive* data from external actors and output ports *send* data to external actors. These port receive and send operations must be identified within the actor DFG. In order to combine these actor SDF graphs into the complete SDF top-level model, the actor ports must be identified in the actor dataflow graph.

For convenience, the production or consumption of a token is referred to as a *port event*. In a Java program, the port events are explicitly ordered by the order they appear in the code. The ordering is lost when converting the method to a DFG, since the nodes representing port events will not be connected. For ports that have multiple port events per actor firing, the order of the events must be determined so the port event nodes can be connected in a way consistent with the semantics of the original SDF model.

Each port event node will be annotated with a port event number, which represents the ordering of the port events. For example, a port that consumes two tokens in a firing will have (at least) two port event nodes in the DFG. Each node will be annotated with a "1" or a "2" to indicate that the nodes denotes the first or second token consumed. Then at the top level the port that produces the token that should be consumed second will be connected to the port event node annotated with a "2".

The port events are indicated in code by a `send()` or a `get()` call on a port object. The number of calls to these methods must be at least as high as the token production or consumption rate on the port. In other words, an output port with production rate of three must have at least three calls to its `send()` method. The number of method calls could exceed the production or consumption rate if two or more of these calls are in mutually exclusive control paths. In any possible control path, however, there must be exactly as many calls as the corresponding rate.

This can be illustrated with a example:

```
public void fire(){
  int a = input.get(0);

  if (a > THRESHOLD){
    output.send(HIGH_VALUE);
  } else {
    output.send(LOW_VALUE);
  }
}
```

The production rate on the output port is only one, but there are two `send()` calls in the above code. They are located in mutually exclusive control paths, so only one of these `send()` calls will be executed per iteration.

In a homogeneous SDF model, each port can only produce or consume one token per iteration. Every instance of the method call, then, must refer to the same port event, whether production or consumption. In a multi-rate graph, however, it is not immediately known which instance of the method call corresponds to which port event. In an iteration where there are four method calls, but the token rate on the port is only three, it must be determined which method call corresponds to which port event.

For example, consider the following code:

```
  public void fire(){
    int a = input.get(0);

    if (a == SOME_VALUE)
1)    output.send(a * 2);
    else
2)    output.send(a * 3);

    if (a > THRESHOLD)
3)    output.send(HIGH_VALUE);
    else
4)    output.send(LOW_VALUE);

5)  output.send(a + 1);
  }
```

There are five `send` method calls, but a quick examination reveals that only three can execute in one iteration, so the port must have a token rate of three. Each method call will be a node in the final dataflow graph. These nodes need to be annotated with the port event number they represent so they can be properly connected at the top level of the model. In this example, the nodes for lines (1) and (2) would be annotated with a one, the nodes for lines (3) and (4) would be annotated with a two, and the node for line (5) would be annotated with a three.

The topology of the SDF model will determine how these ports are to be connected to other actors. These annotations are important so that at the model level, the token that should be consumed second in an iteration will be connected to the second invocation of the port.get() method for the actor. This will preserve the semantics of the SDF model.

To make the port event annotations, each path in the CFG will be analyzed. For each path, the CFG is traversed and the dataflow nodes of the CFG are examined for port events. As the method calls for port events are encountered, the corresponding node in the DFG will be annotated. A counter is kept for each port, and incremented whenever a port event is reached. The counter value is annotated on the DFG node. Some method calls may be encountered in many different control flow paths, but they will represent the same port event in each path.

Figure 3.11 shows the final DFG for the behavior of the SimpleFIR actor. The annotations have been added for the ports. Two delay elements are also shown for the state variables `delay1` and `delay2`. Two mux nodes are inserted, which are used to resolve the definition in multiple paths of the `mac` value. There are five operation nodes which show the calculation of `mac`, and two comparator nodes to perform the clipping.

3.7 Limitations

While this approach can generate dataflow graphs from Ptolemy actors described in Java, there are a number of notable limitations. There are a number of Java constructs that cannot be processed by the current technique.

The primary limitation of this approach is that it does not support any cyclic control flow programming constructs (i.e., loops or feedback jump statements). The technique for merging mutually exclusive control flow paths into a single dataflow graph assumes that the control flow graph is acyclic. Some loops, with static loop counts, can be converted to acyclic control constructs by using well-known loop unrolling techniques.

Another limitation of this technique is that it does not support the use of method invocation within an action method. Although the use of certain methods are allowed (i.e. port `get` and `send` methods, and `Token` class operation methods) to identify known behavior within the control flow graph, the behavior of arbitrary method calls are not included in the graph. Fortunately, Soot provides a mechanism for code inlining to provide rudimentary support for method invocation. Recursion and other unbounded method invocation techniques cannot be supported using this approach.

The last major limitation of this system is that it is limited to scalar data types and certain kinds of arrays. Extracting static dataflow from methods using array types is not possible unless the array indexes can be statically computed during program analysis. A static array index analysis technique will need to be applied to provide rudimentary array data type support, which has not been added yet.

3.8 Conclusion

This chapter has demonstrated how to extract a DFG representation for the behavior of a Ptolemy actor described with Java code. This DFG has the same semantics as the code, but parallelism has been revealed so that a fast hardware implementation can be made.

The original SDF model is now hierarchical: the nodes of the synchronous dataflow model are themselves dataflow graphs. This hierarchical model will be flattened to a more convenient form for representing hardware. The next chapter will show how these DFG representations of individual actors can be combined using the SDF model topology to make a single abstract circuit graph.

This datflow extraction tool has been integrated into Copernicus, the code-generation tool for Ptolemy. The SimpleFIR example shown here was produced with Copernicus. In addition, other sample Java actors have been synthesized using this technique. A discrete differencing actor, which outputs the difference of the two most recent inputs, and an averaging actor, which outputs the running average of a sequence of input values, have been synthesized. These examples have been manually inspected to verify there are correct descriptions of the intended Java behavior.

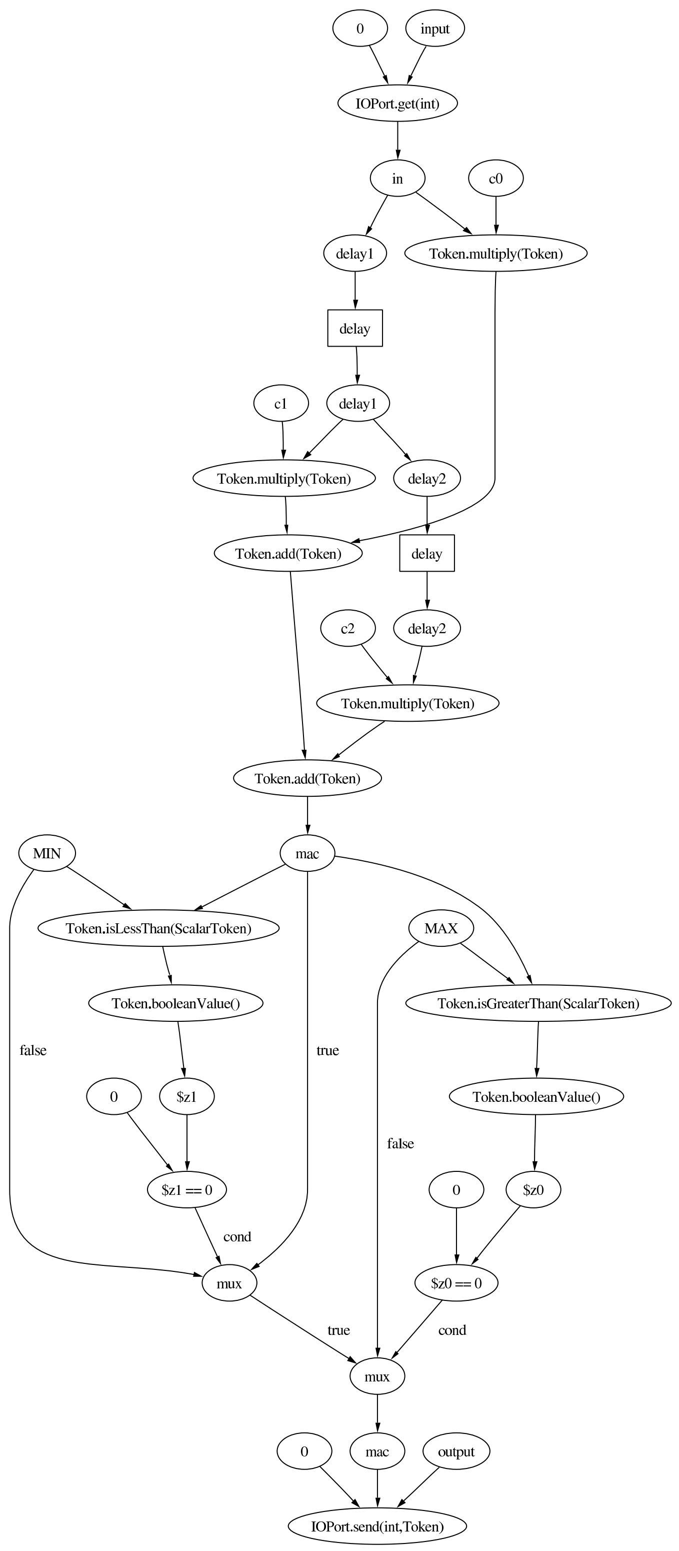

Figure 3.11: Full dataflow description for the SimpleFIR actor.

Chapter 4

Creating Abstract Circuit Graphs

The next step in this synthesis process is to convert the hierarchical SDF graph into a form that can easily be translated into a hardware description. This chapter will describe the process of converting the hierarchical SDF graph into a abstract hardware representation called the abstract circuit graph or ACG. This chapter will begin by introducing and motivating the use of the abstract circuit graph. The first step in this process is to place SDF actor nodes into the ACG. This may involve actor replication for multi-rate actors. The next major step is to replicate actor ports and connect them according to the semantics of the original SDF graph. The last step of this process is to replace the SDF actor node with its dataflow representation as created in Chapter 3. As part of placing the dataflow representation in the ACG, actor state will need to be modified to maintain the same semantics as the original SDF model.

4.1 Abstract Circuit Graph

The goal of this synthesis approach is to generate a circuit with the fastest implementation by exploiting the most parallelism possible. This implies using dedicated resources for each operation so that an entire iteration can execute in a single hardware cycle. However, a multi-rate SDF graph does not naturally lend itself for execution of a single iteration within a clock cycle. In a multi-rate SDF graph, actors may fire multiple times in an iteration. A hardware implementation of a multi-rate actor requires either multiple clock cycles to execute or multiple copies of the actor. Because this synthesis approach will generate hardware that executes a single iteration in a single clock cycle, the latter option is taken. In order to achieve this goal, this phase of the synthesis process will replicate actors to create a single clock cycle iteration.

A new representation of the SDF model will be created in which actors are fired only one time per iteration and produce or consume one data token on each port. This representation of the SDF model can be easily mapped to a hardware implementation that executes an iteration of the model in a single cycle. This representation is called the abstract circuit graph or ACG. Nodes in the final ACG represent a hardware unit, such as an adder, multiplier, comparator, registers, etc. The arcs of the ACG are wires connecting the ports of the hardware units. The ACG is a flat graph of primitive hardware operations that can be easily mapped to a number of hardware technologies.

The abstract circuit graph is very similar to the homogeneous SDF graph described in Chapter 2. Each port in an ACG will always produce or consume a single token per iteration. The arcs in an ACG are the datapaths that the tokens will follow, and show the data dependencies between nodes. Each node in an ACG will only fire once per iteration, just as in a homogeneous SDF graph.

The difference between an ACG and a homogeneous SDF graph lies in the handling of delays. In an SDF graph, delays are shown by initial tokens on arcs between actors. In an ACG, a delay is represented by an actual node in the graph. This delay node will consume a token on its input port and produce that same token on its output port one iteration later. Since all ports must produce or consume a token on every iteration, the delay node must have an initial token to produce during the first iteration. Semantics of the delay are the same, but the representation of delays as nodes more closely resembles an actual hardware implementation. Showing delays as nodes also assists in the algorithm presented in this chapter.

The ACG has been designed to closely resemble the semantics of hardware. The arcs are like wires, which transmit values but cannot store them. The nodes are like simple, combinational hardware execution units, which simply read the value present on their inputs and some propagation delay later drive the appropriate value on their outputs. The nodes that represent delays are very much like hardware registers, which delay for one clock cycle the value from the input to the output.

4.2 Replicating SDF Actors

The actors in an ACG only fire once per iteration. The first step to create an ACG from an SDF model is to replicate the actors once for each time they are fired in an iteration. These replicated actors will each fire once per iteration.

The firing vector for the SDF model determines how often how an actor fires in an iteration. Figure 4.1 shows the graph that will be used as an example to illustrate the process. A balanced iteration of this graph requires that actor A fire three times, and actors B, C, and D each fire twice. Note that a full schedule (i.e., an ordering of the firings in the firing vector) is not needed for this algorithm, only the number of firings.

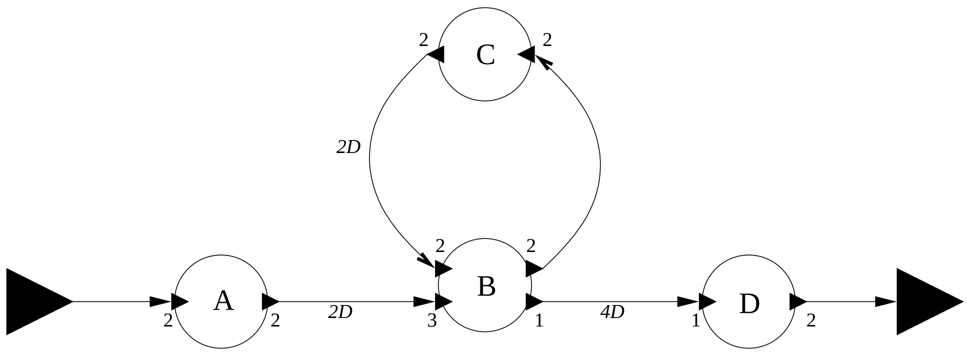

Figure 4.1: Sample multi-rate SDF top level model. The firing vector is: $3A2B2C2D$.

Figure 4.2 shows the ACG with the actors replicated. Actor A fires three times, and it has three instances in the ACG. Likewise actors B, C, and D each fire twice and so have two instances each in the ACG.

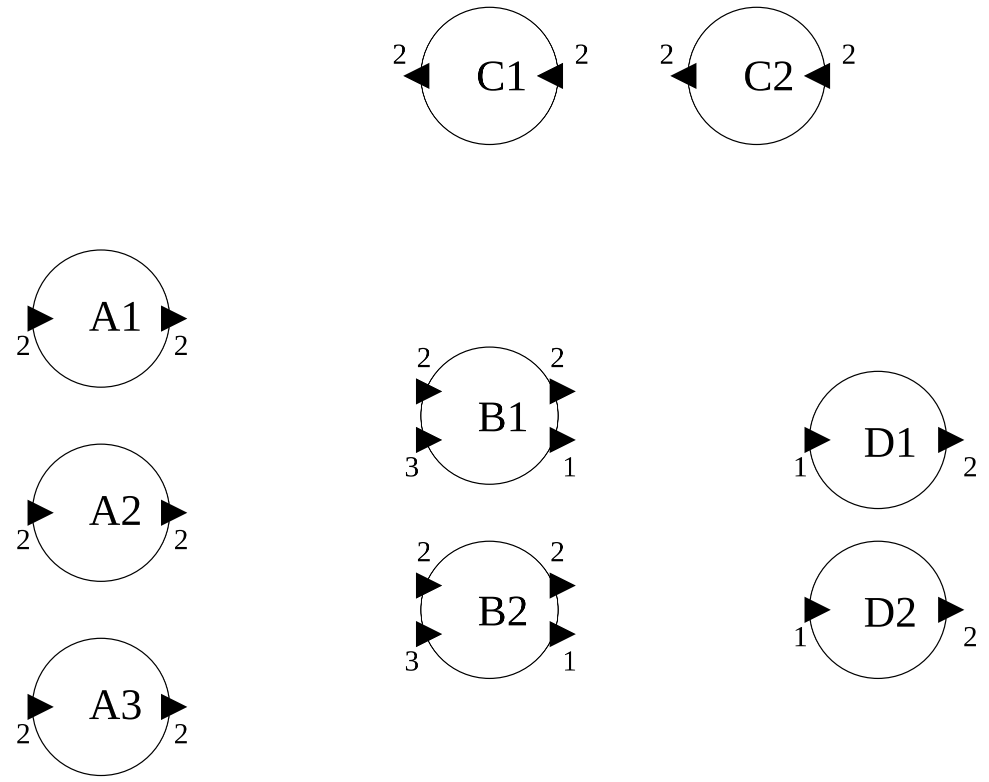

Figure 4.2: ACG with replicated actors.

The problem with is this graph is that it is not immediately apparent how the ports should be connected so as to retain the original behavior of the SDF model. There are now three output ports for actor A nodes, but only two corresponding input ports for actor B nodes. Also, there are initial tokens on the arcs A→B, B→D, and C→B in the SDF model. These initial tokens must be accounted for when connecting the replicated actors.

4.3 Port Mapping

Port mapping is the process of determining how to connect the replicated actors such that the ACG has the same behavior as the original SDF model. A port is the interface between the actor and the graph, and it is through the port that an actor receives (or sends) a data value. This mapping is to preserve the semantics of the original SDF model. The first token consumed on the input side must be the first token produced. In an SDF model this is achieved through the use of a FIFO on each arc to queue tokens that are produced until the input port consumes them. In the ACG, though, each arc only represents a single token transmission per iteration. The correct connection must be made between the inputs and outputs to preserve the original semantics.

This would be a trivial mapping except for initial tokens, those tokens that exist on an arc before the output port produces any tokens. These initial tokens will be the first ones consumed, and they represent a delay in the time between a token is produced on the output and it is consumed on the input. For example, in the case of an arc with two initial tokens, the first token produced by the output port will actually be the third token consumed on the input side.

The first step in port mapping is to replicate each port for each token it produces/consumes, so that each port only produces or consumes a single token. In an ACG, all ports are homogeneous since the arcs in an ACG have no storage, unlike an SDF arc. Figure 4.3 shows the ACG with the ports replicated. Notice that whereas there were an unequal number of ports between actors A and B after actor replication, now each actor has the same number: the actor A nodes have six output ports, and the actor B nodes have six corresponding input ports.

In an ACG, initial tokens are represented by delay nodes. It has been mentioned before that initial tokens are equivalent to delays, since they delay the consumption of a token produced as an output. These delay nodes are added to the ACG, yielding the ACG of Figure 4.4.

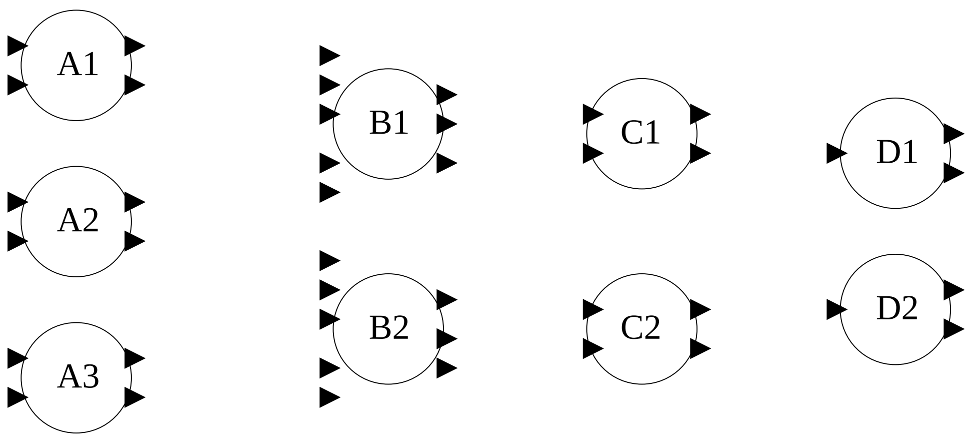

Figure 4.3: ACG with replicated ports.

Next, the single-token ports must be connected to each other. These ports must be connected such that the tokens are produced and consumed in the same order as in the original SDF model. In general, the first token produced will be the first token consumed. However, initial tokens will change this mapping, since all initial tokens must be consumed before any produced tokens can be consumed. The initial tokens must also be replaced, so that at the end of the iteration there are the same number of tokens as before the iteration. This means that the delay nodes must have their inputs connected so they can replace the token they output during an iteration.

Figure 4.5 illustrates how initial tokens affect the mapping between single-token ports on the actors. On the left is a simple SDF model with no initial tokens. The corresponding ACG is shown below it. The arcs are connected straight across: the top-most output port is connected to the top-most input port and so on. The SDF model on the right, however, contains delays, which changes how the ports are connected to each other.

When delays are present, the port representing the first consumed token is connected to the delay node for the first initial token. The second consumed token

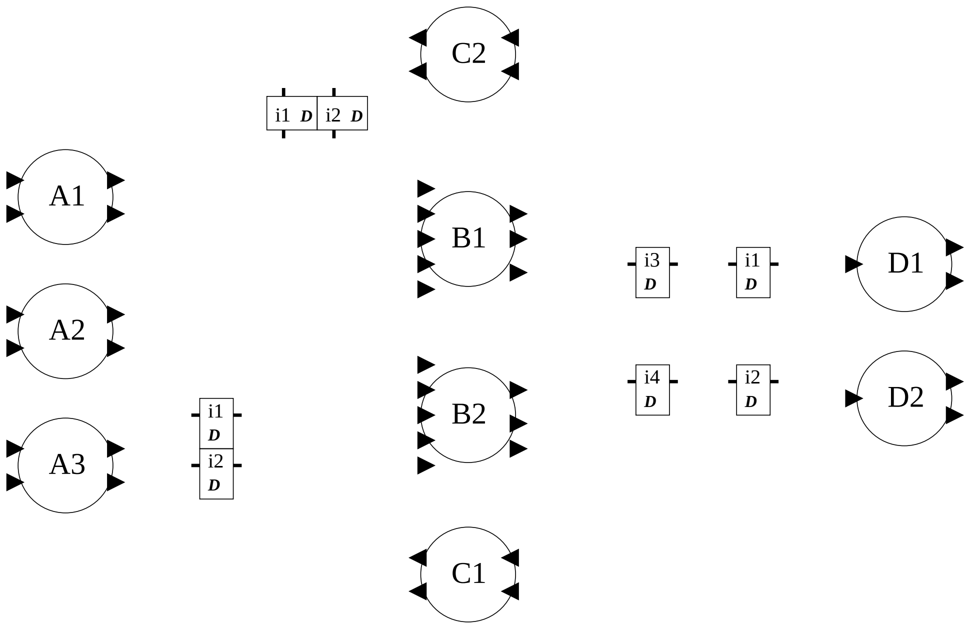

Figure 4.4: ACG with replicated ports and delay nodes.

port is connected to the second delay node, and so on until all delay nodes have their outputs. The next consumed token port (or the first one, if no delays exist), is connected to the first produced token port. These connections are made in like manner until all actor input ports have been connected.

Finally, the delay nodes must connect their input ports. The delay node for the first initial token connects its input to the first unconnected output port. In the case where there are more delays on an arc than tokens consumed in an iteration, there will be more delay nodes than actor input ports. So the first unconnected output port will be another delay node. In this way, a token could be delayed multiple iterations before it is consumed. The second delay node then connects its input port, and so on until all delay nodes have their input ports connected. At this point, all ports will have a connection.

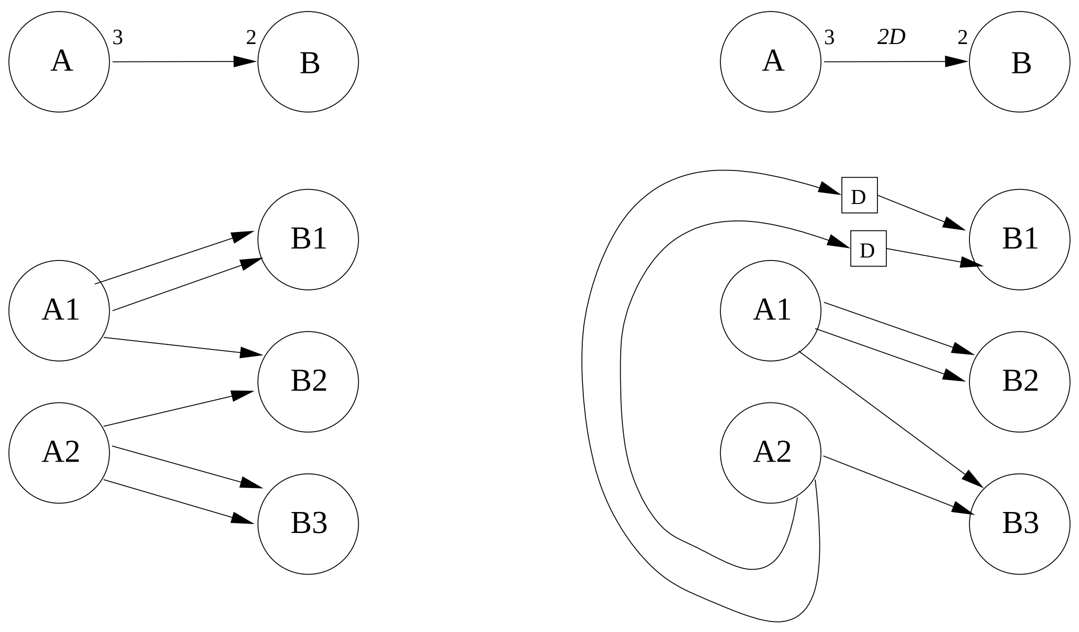

Figure 4.5: ACGs for SDF models with and without initial tokens.

Figure 4.6 shows the transformation up to this point of the multi-rate graph of Figure 4.1 to an ACG. The ports are shown in ascending order from top to bottom on each actor.

4.4 Inserting Actor DFG

At this point, the nodes of the ACG represent actor firings. The DFGs obtained in Chapter 3 for each actor instance are used to replace the node for each actor firing. The actor ports on the nodes will be connected to their equivalent nodes on the actor DFG, using the port ordering annotations (described in Chapter 3). For example, an input port annotated with a one in the actor DFG is the first token consumed by that input port, and will be connected to the corresponding port in the ACG. Likewise the actor output ports will be connected to the actor DFG nodes with the corresponding annotation. The ACG is now a large, flat graph of primitive operations that can be easily translated into a structural circuit description.

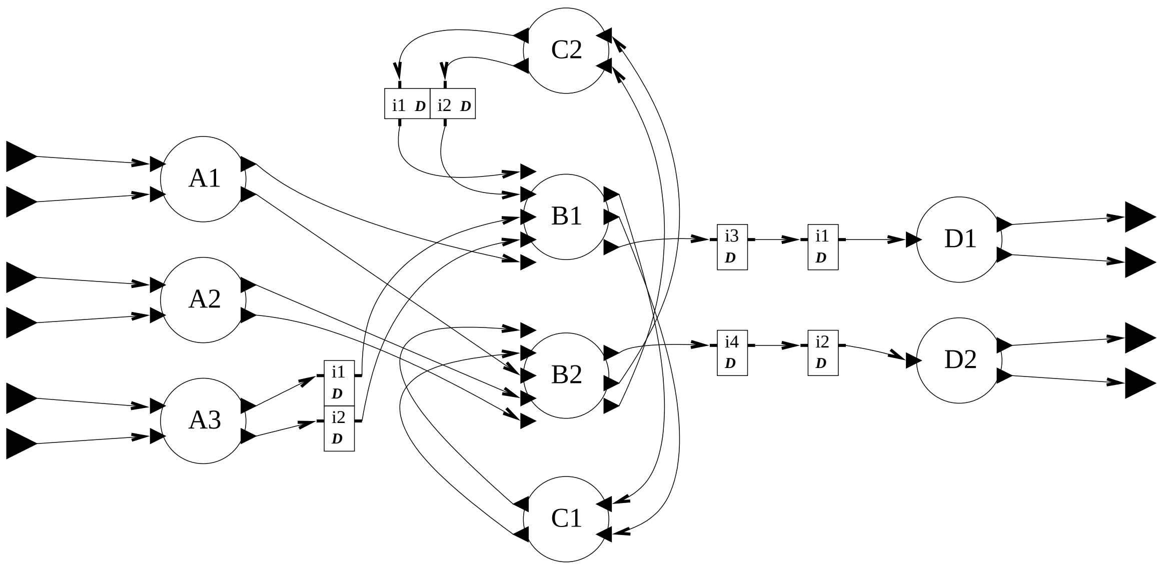

Figure 4.6: The complete homogeneous graph.

The semantics of an actor firing multiple times in one iteration are that the state from the previous firing in the same iteration would be available in the next firing. As shown above, the actor DFG is replicated for each firing in the iteration. This would mean that each firing of an actor would keep its own state, instead of all firings from the same actor sharing the same state. Figure 4.7 shows this situation for actor A.

State is any persistent data value that is read, then updated. State is identified during the behavioral DFG extraction of the Java actor. The state connection should be modified so that the next state values computed from the previous firing become the current state in the next firing.

The next state values computed by the last firing will become the input for a delay element. The delay element's output will be the current state values for the first actor firing. Figure 4.8 shows the correct state connections for actor A. Now the state computed by the first firing of actor A will be read and updated by the second

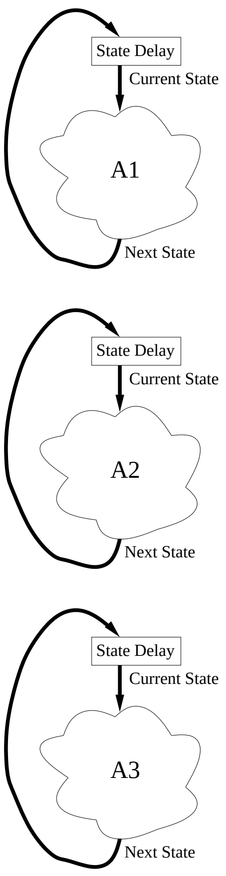

Figure 4.7: Replicated actors each retain their own state.

firing, which will be read and updated by the third firing. The final state values from the third firing will be saved in delay elements, and they will become the input during the next iteration for the first firing.

4.5 Conclusion

This chapter has shown how a general SDF model can be transformed into an abstract circuit graph. This ACG is composed of primitive logic and arithmetic operations. The ACG is designed to be able to run in a single hardware cycle using dedicated resources for maximum speed. At this point, standard graph optimizations

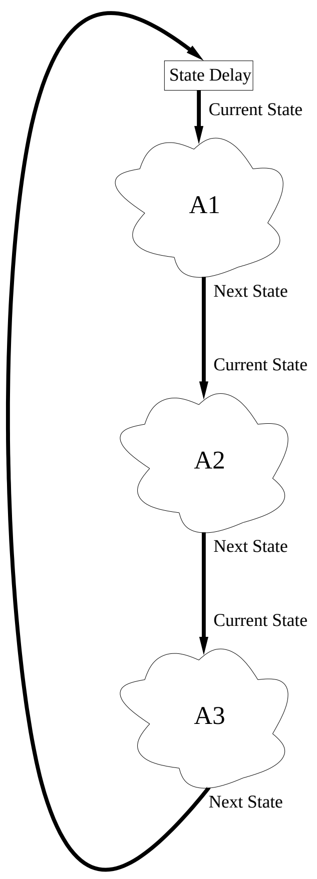

Figure 4.8: Modified state connections to retain original state semantics.

could be employed to increase the speed of the hardware or reduce resource usage. Importantly, the ACG retains the same behavior as the original SDF model, but in this form can be easily translated into a structural circuit description, such as VHDL or Verilog.

Chapter 5

Conclusion

This thesis presents a method for obtaining a hardware representation from an algorithm described using an SDF model where the actors are restricted Java programs. The SDF models that can be synthesized can be either homogeneous or multi-rate, and the Java code that is synthesized must conform to the Ptolemy specification. The hardware representation generated by this technique can readily be transformed into a structural hardware description.

The synthesis approach consists of several major tasks. First, the behavior of the software actors is extracted and represented with a dataflow graph. The SDF model is then converted into an abstract circuit graph, and the dataflow graph representation of each actor is used to replace the node representing that actor in the graph. The end result is a graph whose nodes are primitive operations, and whose nodes all have homogeneous token rates. This graph has semantics which closely resemble a hardware circuit, and could be translated into any number of hardware descriptions using a structural hardware design language.

5.1 Future Work

Further work in this hardware synthesis technique could relax the limitations on actors that are synthesizable. Loop unrolling and method inlining would greatly increase the breadth of code that code be synthesized.

Currently, no graph optimizations are considered for the final SDF graph before it is converted to hardware. Some optimizations could reduce the amount of hardware required or increase the speed. For example, retiming could be used to pipeline the hardware. The user would decide how many pipe stages he wants, and retiming would insert the necessary pipeline registers to minimize cycle time. Also, as described, this process synthesizes hardware which performs the entire computation in a single cycle. A multi-cycle solution could be explored, which would require hardware scheduling but would allow resource sharing for a smaller circuit.

The typing and widths of the data tokens transferred have not yet been considered. All tokens have been assumed to be signed 32 bit integers. While a reasonable starting point, this could lead to a waste of hardware. Some hardware elements, such as multipliers, increase in area exponentially with the size of the inputs. If an algorithm required less precision than 32 bits, then using the minimum possible could greatly reduce the hardware required to implement it. On the other hand, if a calculation requires more than 32 bits of precision, it is currently not representable by this technique. Performing a bit-width analysis on the SDF model would be very useful.

Also, typing could be performed, for example to tell the difference between integer and floating point types. Floating point types would require special floating point hardware to preserve the semantics of the actor. Currently no mechanism is in place to determine the types of tokens, so adding this feature could enable a wider range of actor code.

5.2 Concluding Remarks

Modeling algorithms using a dataflow graph provides a natural and convenient way to express algorithms that are dominated by data and require very little control flow. These dataflow graphs show operations and their data dependencies only, so that maximum parallelism is revealed by this approach. Many research projects have explored ways to implement these dataflow descriptions, both in software and in hardware.

Software implementations are flexible, allowing the designer to specify custom actor behavior easily. They can be slow to execute, however. Hardware implementations are usually faster, but the design process takes longer. Designers are usually limited to predefined actors for which implementations exist in the target technology.

One form of dataflow particularly useful for synthesis is synchronous dataflow. The distinguishing feature of SDF is that rate of data production and consumption for each node is fixed. Analysis of these types of dataflow graphs can be performed at compile time.

This thesis has demonstrated a way to combine the benefits of both. Arbitrary actors are allowed, subject to some software constraints. Also, the final representation after synthesis allows the model to be executed in hardware.

Appendix A

Labeling Algorithm

An important part of the interval analysis discussed in Chapter 3 is to find the exit points for intervals. The interval exit points are identified through a system of labeling. A label is a designation created every time a branch in execution is reached. Labels are "destroyed" whenever the block controlled by the branch ends. Thus labels mark the beginning and end of intervals.

When an interval begins, two labels are created which are propagated down each possible execution path. When the labels both come into the same node, the interval marked by these labels must be ending, so a JOIN node is placed in front of that node.

The main purpose of this algorithm is to determine where the interval ends. This will be accomplished in two steps: matching labels and combining paths. These steps will reduce the execution paths entering a node to a single one. The labels on this remaining path will then be propagated to this node's successors. The next node in the topological sort is selected, and the process repeats.

A label queue is a LIFO (last-in, first-out) queue of labels. A queue is associated with each arc of the graph. When a new condition is reached, a new label is pushed onto the label queue. The label contains the information about which condition created it, and which branch (true or false) of the condition this path represents. When a label is matched, it is popped off, exposing the next most recent entry. So a label queue represents which conditions were encountered to reach this path, and what those conditions must resolve to for this path to have executed.

The nodes of the CFG are processed in topological order. A topological order only exists in directed, acyclic graphs, which is the reason that feedback control(e.g., loops) is not allowed. The topological order is necessary since this algorithm requires that all predecessor nodes be processed before the current node.

A.1 Label Matching

To reduce execution paths, the labels must be matched. When a node is processed, the labels queue from every incoming path is compared to the label queue in every other incoming path. The most recent label from each label queue is pairwise compared, and if the conditions that created these labels are the same, the two labels are said to match.

A matched label signals the end of an interval. Both execution branches of a condition are merging into the same node, so the condition no longer applies. The end of the interval is marked by a JOIN node.

The condition port determines whether the true or false input is propagated. The `if` condition (which is a Boolean expression) that created these labels is connected to the condition port of the join node. The label on the output node is the common part from the two merged labels. The merged labels are then deleted from their corresponding paths. If a path contains no more labels because of the deletion, then the path is also deleted.

Matching then starts over, as the newly created path and label could be eligible to match again with another label. This process continues until every label has been pairwise compared with no matches, or only one incoming path remains.

Example Figure A.1 shows the control flow subgraph for a typical if-then-else statement. Node A contains the `if` condition. Node B is the `then` clause, executed when the `if` evaluates to true, while node C is executed if the condition is false. Node D is the first basic block after the entire `if` block.

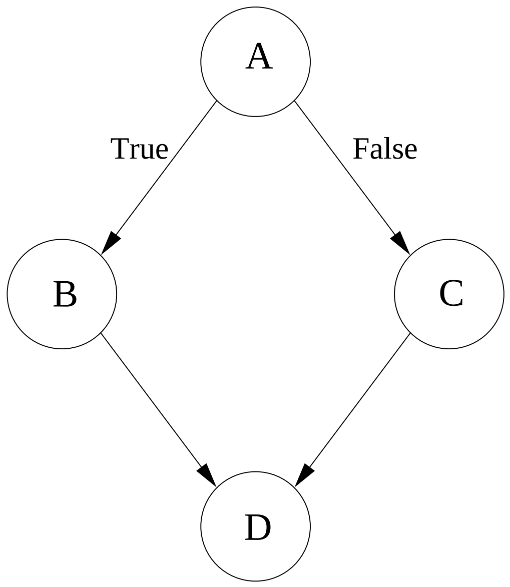

Figure A.1: Control-flow subgraph for typical If/Then/Else statement.

Figure A.2 shows how the labeling would look as node D begins to be processed. As node D begins, it has two incoming paths, and each path has one label queue each (also each queue only has one label in it). It tries to match the labels by comparing their most recent label in each queue. The labels were created by the same condition (the condition that ended Node A), so these labels match. A JOIN node (see Figure A.3) is created, and the arc from B is attached to the true input of the JOIN, while the arc from node C is connected to the false input. The condition that will determine which input is propagated through the JOIN is the condition that created these two labels in the first place. In this case, the condition is found in node A. So an arc, representing the evaluation of the `if` conditional is made from node A to the new JOIN node.

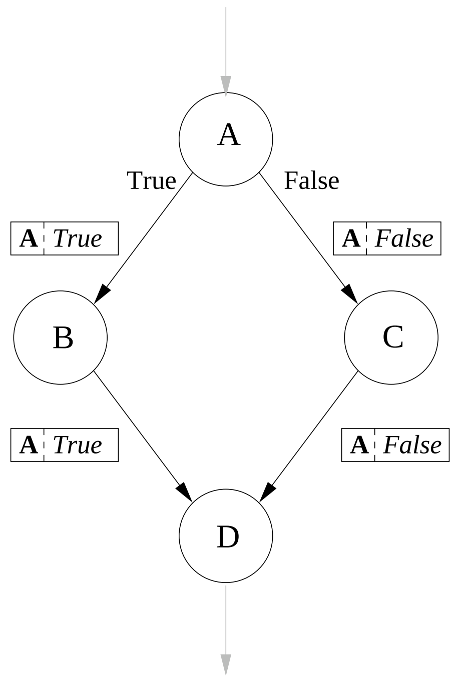

Figure A.2: Control-flow subgraph with labels for typical If/Then/Else statement.

The output of the JOIN node is connected as an incoming path to Node D (see Figure A.3). If two labels match, then their corresponding queues will be the same as well (except for the last entry). So the matching labels are popped off, and the common part of the queue will become the new label for the output of the JOIN node. Now, since only one path is left entering Node D, matching terminates.

A.2 Propagate Labels

When there is a single execution path entering this node, the label queue on that path has an entry for every condition that is in effect, and the value the condition

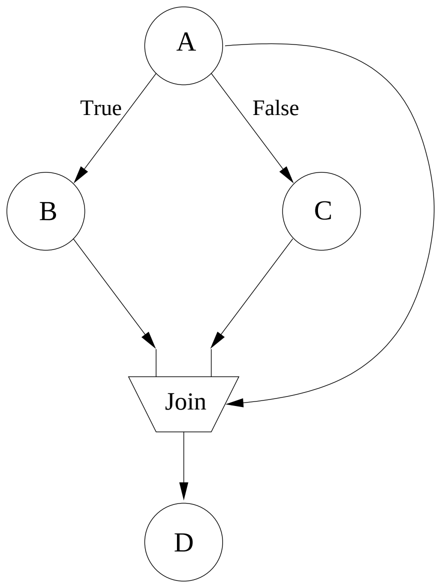

Figure A.3: Typical If/Then/Else after merging.

must be for this path to execute. The outgoing execution path(s) from this node will likewise need to contain the information to indicate whether those paths are executed. The label queue from the input execution path is propagated to the outgoing paths, and the queues are updated to show the conditions in effect upon exiting this node.

Two cases are possible when leaving a node: the control flow could branch due to a new condition being reached, or control flow could proceed unambiguously to another node. If this node has only one outgoing path, then the conditions that were in effect upon entering this node are the same upon exiting. If the node has two outgoing paths, then a branch has been reached. All the previous conditions are still in effect, but now a new entry must be added to the label queues to reflect this new branch.

If the current node has only one successor, then no control flow change occurs at this node. Since no new conditions are reached, the label queue on the input path is copied to the output path. Figure A.4 shows the case where the node has a single successor.

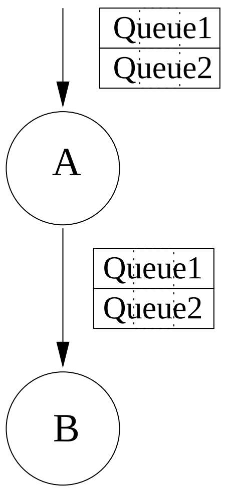

Figure A.4: Node A is the current node. Since it has only one successor, the labels from the input path are copied unchanged to the output path.

If the node has two successors, then an `if` statement must end the basic block represented by the current node, so a condition has been reached. Two copies of the label queue in the incoming path are made. A new entry is then pushed onto the new label queues indicating this condition as the creating condition. The label queues are added to the outgoing arcs, with the latest entry in each queue reflecting whether it is the true or false path from the creating condition. Figure A.5 shows labels splitting at a branch.

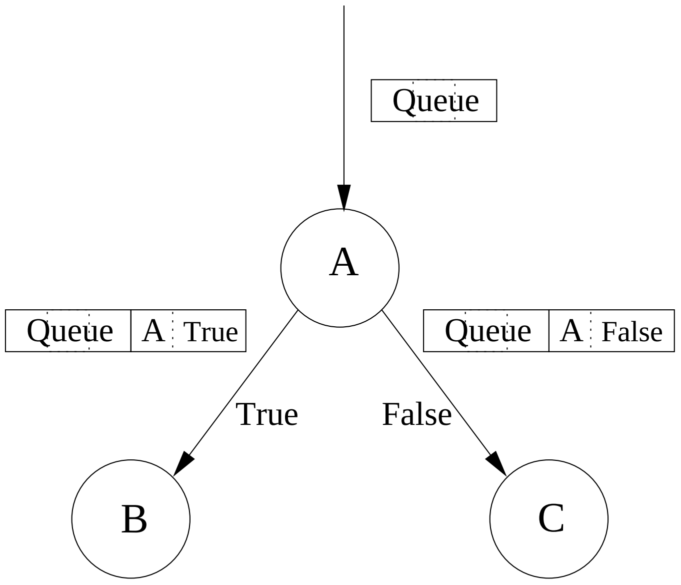

Figure A.5: Node A is the current node. Since there is a branch in Node A, the label from the input path is copied to each output path and extended with information about the condition in Node A.

Bibliography

[1] J. B. Dennis, “Data flow supercomputers”, *Computer*, vol. 13, no. 11, pp. 48–56, November 1980.

[2] A. L. Davis and R. M. Keller, “Data flow program graphs”, *Computer*, vol. 15, no. 2, pp. 26–41, February 1982.

[3] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow”, *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[4] P.Y. Calland, A. Darte, and Y. Robert, “Circuit retiming applied to decomposed software pipelining”, *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, pp. 24–35, 1998.

[5] C. Leiserson and J. Saxe, “Retiming synchronous circuitry”, *Algorithmica*, vol. 6, pp. 5–35, 1991.

[6] C. Worth, “Hardware compilation of java class files: A synthesis framework for the JHDL CAD suite”, Master’s thesis, Brigham Young University, August 2000.

[7] J. Tripp, P. Jackson, and B. Hutchings, “Sea Cucumber: A synthesizing compiler for FPGAs”, in *FPL*, 2002.

[8] P. Bellows and B. L. Hutchings, “JHDL - an HDL for reconfigurable systems”, in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1998, pp. 175–184.

[9] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using ptolemy", *Journal on VLSI Signal Processing*, vol. 9, no. 1, pp. 7–21, January 1995.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing", *IEEE Transactions on Computers*, January 1987.

[11] P. K. Murthy and S. S. Bhattacharyya, "Shared memory implementations of synchronous dataflow specifications", in *Proceedings of the Design, Automation and Test in Europe Conference*, 2000, pp. 404–410.

[12] P. K. Murthy S. S. Bhattacharyya and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications", *Journal of VLSI Signal Processing Systems*, vol. 21, no. 2, pp. 151–166, June 1999.

[13] S. Neuendorffer, "Automatic specialization of actor-oriented models in Ptolemy II", Master's thesis, University of California at Berkeley, December 2002, Technical Memorandum UCB ERL M02/41.

[14] M. C. Williamson, *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*, PhD thesis, University of California, Berkeley, Spring 1998.

[15] H. Jung, K. Lee, and S. Ha, "Efficient hardware controller synthesis for synchronous dataflow graph in system level design", in *International Symposium on System Synthesis, 2000. Proceedings, 13th*, September 2000, pp. 79–84.

[16] E. A. Lee, "Overview of the Ptolemy project", Tech. Rep. Technical Memorandum UCB/ERL M01/11, EECS, University of California, Berkeley, March 6 2001.

[17] M. Adé, R. Lauwereins, and J. A. Peperstraete, "Hardware-software codesign with GRAPE", in *6th IEEE International Workshop on Rapid System Prototyping*. IEEE, June 1995, pp. 40–47.

[18] R. Lauwereins, M. Engels, M. Adé, and J. A. Peperstraete, "Grape-II: A system-level prototyping environment for DSP applications", *Computer*, vol. 28, no. 2, pp. 35–43, February 1995.

[19] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems", *International Journal of Computer Simulation*, vol. 4, pp. 155–182, April 1994.

[20] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. L., X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong, "Heterogeneous concurrent modeling and design in Java", Tech. Rep. Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, March 15 2001.

[21] E.B. Hogenauer, "An economical class of digital filters for decimation and interpolation", *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 29, no. 2, pp. 155–162, 1981.

[22] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification(2nd Edition)*, Addison-Wesley, 1999.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a Java optimization framework", in *Proceedings of CASCON 1999*, 1999, pp. 125–135, http://www.sable.mcgill.ca/soot.

[24] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[25] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock", in *25th Annual International Symposium on Microarchitecture*, 1992.

[26] W. W. Hwu, S. A. MahIke, W. Y. Chen, P. P. Chang, N. J. Water, R. A. Bringmann, R. G. OueIIette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Hohn, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation", in *Journal of Supercomputing*, January 1993.

[27] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Predicated static single assignment", in *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[28] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow", in *Symposium of Principles of Programming Languages*, 1995, pp. 37–48.

[29] G. Agrawal, "Simultaneous demand-driven data-flow and call graph analysis", in *Proceedings of the IEEE International Conference on Software Maintenance*, 1999, pp. 453–462.

[30] X. Yuan, R. Gupta, and R. G. Melhem, "Demand-driven data flow analysis for communication optimization", *Parallel Processing Letters*, vol. 7, no. 4, pp. 359–370, 1997.