LYRA : A HIGH LEVEL MODELING AND SYNTHESIS METHODOLOGY FOR CONCURRENT SYSTEMS USING RENDEZVOUS

VYAS VENKATARAMAN

Dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

LYRA : A HIGH LEVEL MODELING AND SYNTHESIS METHODOLOGY FOR CONCURRENT SYSTEMS USING RENDEZVOUS

by

VYAS VENKATARAMAN

M.S., Boston University, 2008

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2012

Approved by

First Reader

Martin Herbordt, PhD
Associate Professor of Electrical and
Computer Engineering

Second Reader

Wei Qin, PhD
Senior Software Developer,
Tower Research Capital, LLC

Third Reader

Jayanta Bhadra, PhD
Manager, Verification Tools and Flows,
Freescale Semiconductor Inc.

Fourth Reader

Ayse Coskun, PhD
Assistant Professor of Electrical and
Computer Engineering

Acknowledgments

I would like to thank Dr. Alexander Taubin for sparking my interest in the area of digital hardware design and for his guidance. My thanks also to Dr. Wei Qin for introducing me to the field of communication abstraction and co design and for his mentorship. Thanks also to Dr. Martin Herbordt for his patience and insight into polishing and improving this work, and Dr. Jayanta Bhadra for his industry related feedback that allowed this work to be more relevant. Additionally, thanks to Dr. Ayse Coskun for serving on the defense and Dr. Doug Densmore for his valuable counsel. I would like to thank the help of the ECE office staff, particularly Julie Guthrie.

Finally, I would like to thank my parents and my wife for their constant patience, generosity and unyielding support. Their presence and support has been instrumental throughout the PhD program.

LYRA : A HIGH LEVEL MODELING AND SYNTHESIS METHODOLOGY FOR CONCURRENT SYSTEMS USING RENDEZVOUS

(Order No.)

VYAS VENKATARAMAN

Boston University, College of Engineering, 2012

Major Professor: Martin Herbordt, PhD,
Associate Professor of Electrical andComputer Engineering

ABSTRACT

Since their introduction, modern computer systems have been increasing in complexity. System designers have been dealing with ever larger designs by moving to higher abstraction level system descriptions. The existing register transfer level of abstraction has become unable to handle modern designs, requiring a move to high level modeling. The most popular of the current approaches for high level design is using SystemC, a set of libraries built in C++, to model hardware using software concepts. However, software based approaches suffer from a major drawback - the lack of a formal definition for both communication and computation. Basic hardware primitives such as concurrency and multiparty communication cannot be easily expressed in software and the translation of these models into hardware equivalents is difficult. As a result, most designers choose to represent systems as a high level SystemC model for simulation, and a set of register transfer level designs for implementation. This gap presents challenges in the design and verification of the system.

This work proposes a novel high level modeling methodology, called Lyra, which

uses the well studied concepts of finite state machines for computation and of rendezvous for communication. A rendezvous is a bidirectional, atomic, synchronous communication construct that supports a wide variety of communication patterns such as multiparty and variable party communication. The presence of a novel mechanism to handle nondeterminism from the use of rendezvous allows Lyra to model designs that existing rendezvous based approaches cannot. Finite state machine based modeling makes Lyra amenable to hardware implementation and easily understandable by hardware engineers. The formal foundation of Lyra and the ability to implement models as hardware are advantages compared to other high level modeling approaches.

This thesis presents Lyra and the novel rendezvous nondeterminism resolution mechanism, called the communication scheduler. It develops a graph based method to analyze and compare different rendezvous based approaches. This work demonstrates the implementation of Lyra into a simulator and a synthesis tool, creating a practical design flow. This work examines some system models that demonstrate the benefits of using Lyra.

Contents

List of Tables

List of Figures

List of Abbreviations

CAD	.....	Computer-Aided Design
CCS	.....	Calculus of Communication Systems
CSP	.....	Communicating Sequential Processes
DFA	.....	Deterministic Finite Automaton
DME	.....	Deterministically Mutually Exclusive
EDA	.....	Electronic Design Automation
EFSM	.....	Extended Finite State Machine
FSM	.....	Finite State Machine
GTL	.....	Gate Transfer Level
GWO	.....	Global Weight Optimal
HDL	.....	Hardware Design Language
HLS	.....	High Level Synthesis
MCS	.....	Minimal Candidate Schedule
ME	.....	Mutual Exclusion / Mutually Exclusive
NFA	.....	Nondeterministic Finite State Machine
NME	.....	Nondeterministically Mutually Exclusive
ORG	.....	Occurrence Relationship Graph
RTL	.....	Register Transfer Level
SLWO	.....	Static Local Weight Ordering
TE	.....	Transition Edge
TLM	.....	Transaction Level Modeling
TRG	.....	Transition Relationship Graph

Chapter 1

Introduction

Any digital hardware design can be partitioned into two components : computation and communication. Computation refers to the portion of the hardware design dedicated to the transformation of input data to output. The communication component refers to the transfer and synchronization of data between the different computation components. With the increase in complexity of modern hardware systems, there has been a corresponding increase in the complexity of their descriptions. This rising complexity has been contained by simultaneously increasing the level of abstraction of the basic units. By dealing with more abstract representations, larger designs can be modeled and created. While the level of abstraction of computation has been rising over time, the same has not been the case for communication. As a result,in the design of modern digital hardware systems, communication abstraction is increasingly important.

This chapter contains a brief overview of the general concepts in modern digital hardware design, followed by a description of design automation tools and an overview of hardware synthesis. We demonstrate the increasing importance of communication abstraction in behavioral models of digital hardware and briefly describe the solution proposed in this dissertation called Lyra.

1.1 Introduction to Electronic Design Automation (EDA)

Digital electronic circuits, based on the switching activity of transistors are capable of performing complex calculations. When the first integrated circuit (IC), demonstrated in 1958, it could accommodate only a few transistors on it, and could be entirely designed manually. The complexity of modern digital electronics has far exceeded the scale at which manual design of transistors is possible. In 2011, the 32nm Intel i7 processor, for example, contains over 1.17 billion transistors [Kurd et al., 2010]. In order to handle the design and implementation of such large digital hardware designs, automated tools were developed that comprise the Electronic Design Automation (EDA) field.

The representations of a digital system can be divided into 3 main domains : the physical, the structural and the behavioral [Gajski et al., 1992] . Within each domain, we have corresponding *levels of abstraction* forming a hierarchy from simpler to more complex, with each succeeding level using the previous one as its basic element (Figure 1·1). The most basic digital hardware design element, at the lowest level of abstraction, is the transistor [Gajski and Ramachandran, 1994]. In the physical domain, the transistor is represented simply by a transistor layout. A *layout* is actual physical shape of the transistor as created in silicon. In the structural domain, the representation is as a symbol. In the behavioral domain, the representation of the transistor is the "on-off" electrical characteristics of the transistor. A transistor can be equivalently expressed in these 3 domains, with the choice of the domain being the use of the design. For a fabrication facility, which creates physical chips, the most relevant representation is the layout. When identifying connections in a digital system, the most useful representation is the structural. When modeling a system, engineers represent the transistor in its behavioral form.

Moving to the next level of abstraction, we have the gate level model, sometimes

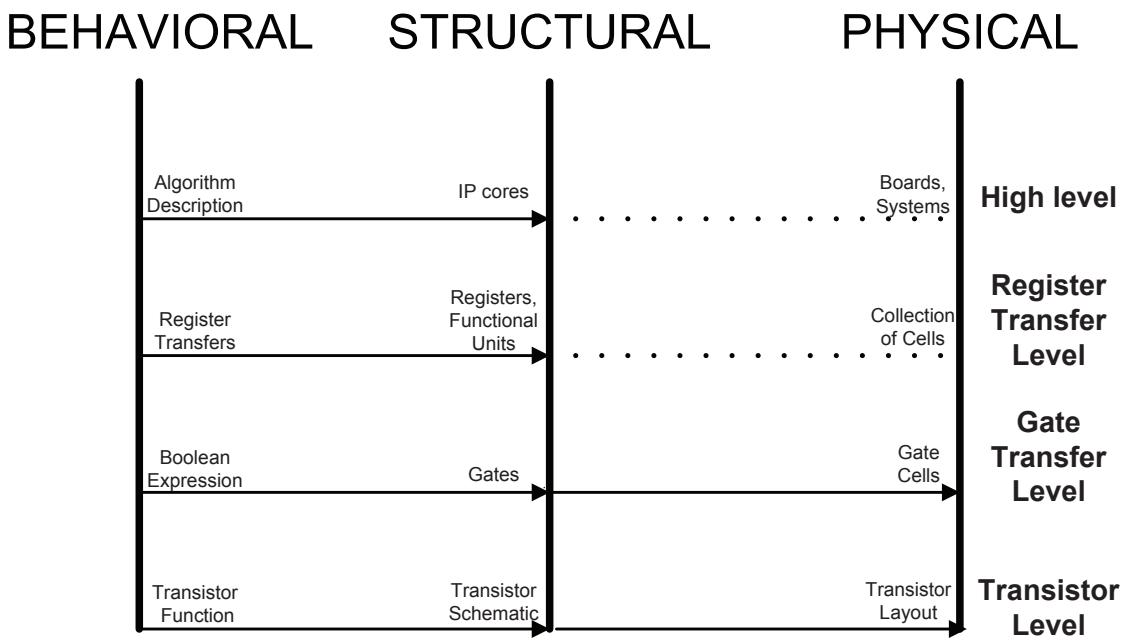

Figure 1·1: The three domains of design at different abstraction levels. Electronic design automation tools for digital hardware accept a description in the behavioral domain and finally transform them into the physical domain. *Synthesis* is transformation from the behavioral to the structural domain. *Refinement* is the transformation within a domain from a higher level of abstraction to a lower one. *Optimization* is the transformation within a domain at the same level of abstraction, but producing a description that is improved for some target criterion. This is based on the Gajski Kuhn Y chart [Gajski et al., 1992]

called the Gate Transfer Level(GTL). This corresponds to layout cells in the physical domain, to gate schematics in the structural domain and to the boolean expression in the behavioral domain. In each domain, this level can be thought of as replacing a collection of transistor level representations. Thus, a layout cell contains the layouts for multiple transistors. The gate schematic is a representation of multiple transistor schematics. The boolean expression represnts the behavior of multiple transistors.

For higher levels of abstraction, the tight correspondence between the various domains weakens. Higher abstraction levels do not always require the introduction of new elements in the physical and structural domains. The descriptions of gates usually occurs as a *netlist*, a structural list of gates and their connection.

The next level of abstraction, the register transfer level (RTL) can best be described as a collection of gates that feed registers. The correspondence in the physical and structural domains is to collection of cells and a collection of combinational gates and clocked state holding elements respectively. In the behavioral domain, the correspondence is to a new style of design where data is processed sequentially in time. This new design style raises the level of abstraction by breaking apart the design into smaller elements that can be designed in parallel and perform computation in a pipelined fashion. Each piece of the design is combinational, but performs actions on data obtained from and sent to registers. In effect, the RTL system can be described in the behavioral domain using a state chart or as a set of register transfer equations.

Hardware Description Languages (HDLs) such as Verilog [IEEE, 2001] or VHDL [IEEE, 2000] are used to define the behavioral input. These languages differ from software programming languages in their assumptions about concurrency, communication and data type handling. Both HDLs and software programming languages support basic concepts such as modularity. Modularity refers to the ability to wrap elements representing computation and storage into block referred to as a *module*. This allows

for reuse of a piece of a design and hierarchical design, by allowing one module to contain *instances* of other modules, allowing for larger designs. On the other hand, HDLs and software programming languages differ in some basic ways. In software programming languages, which assume sequential execution, concurrency must be defined explicitly. HDLs, on the other hand, have an implicit notion of concurrency with each module independent of the other. The difference in handling concurrency affects the basic communication constructs supported by these approaches. Software programming languages generally use function calls as the basic mechanism of communication. In contrast, HDLs generally use wires as the basic communication primitive. The loose semantic definitions of function calls, and their use of features like recursion, variable length arguments prevent them from being easily converted into a hardware equivalent. Finally, the basic datatypes in HDLs are very different from those in software programming. While some types are common, software programming data types often depend on the architecture of the underlying processor as bit widths are not often defined. HDLs, on the other hand, fix the size of data types rigidly to ensure that they are always easily relateable to hardware. These distinctions are further elaborated in Chapter 2.

At present, the level of abstraction that is being used is the system level [Sangiovanni-Vincentelli, 2003] [Densmore et al., 2006]. Elements at this level correspond to *cores* that frequently are actually presented as a collection of RTL elements in the structural domain. These high level models are usually referred to as Intellectual Property cores (IP cores) [Gajski et al., 2000a] or as *soft cores*. This reflects the fact that that the tasks of modern system designers has become integrating different functional blocks from multiple sources. In the behavioral domain, this level of abstraction corresponds to well defined algorithm implementations as extended communicating state machines. In a sign of the growing complexity of modern com-

puter systems, there is a disconnect between the behavioral and structural domains [Hemani, 2004]. The behavioral model at this level of abstraction is usually modeled using software programming techniques, but as the structural model is RTL, there is some extra effort that needs to be expended by the designer to create both representations.

Increasing the abstraction level has been acknowledged as an essential method used by modern EDA tools to handle increasingly complex systems [Sangiovanni-Vincentelli, 2003] [Hemani, 2004]. The abstraction of computation is the primary focus for the increases in the levels of abstraction [MacMillen et al., 2000]. Abstraction of communication, on the other hand, has become increasingly important as the primary driver for the future of EDA tool design [Hemani, 2004] [Kumar et al., 2002] [Pestana et al., 2004]. We see that computation has moved in abstraction from being modeled by transistors to gates to registers to IP Cores. Communication, on the other hand, has existed as wires in the first three levels of abstraction, with no clear agreement on its abstraction in the high level era. This presents major challenges for the modern era that may have several IP cores communicating with each other, such as the Network-On-Chip style of design [Hemani et al., 2000] [Pestana et al., 2004] [Kumar et al., 2002] [Gerstlauer et al., 2005].

A practical example of the use of abstraction for communication can be demonstrated in Figure 1·2. In the definition of a system on chip, there is a need to model different patterns of communication between IP blocks. In this scenario, there are two clients, simulating processors capable of complex behaviors and two resources, modeling hardware co-processors, such as a cryptographic decoder. The clients are capable of requesting and acquiring resources. The resources are first requested, exclusively assigned to a single client and then released. The first client attempts to request and use both resources. The second client requests just the first resource.

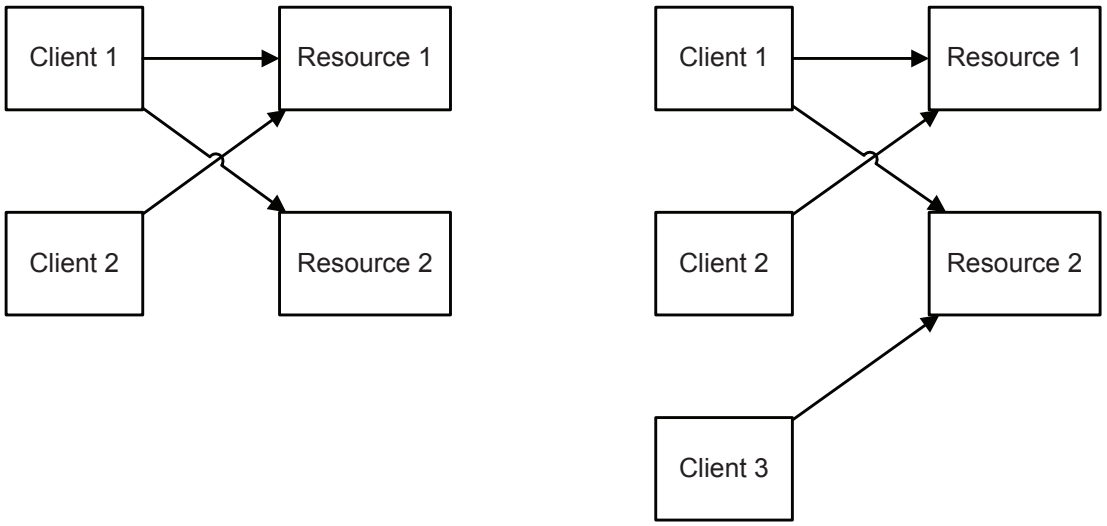

Figure 1·2: A sample of a scenario that demonstrates the need for communication flexibility. If we have a system-on-chip where there are two clients that can request and exclusively use the resources (shown on the left), the abstraction of communication can allow for easy modeling of scenarios, such as the addition of a client (shown on the right)

At any time, one, or both clients could attempt to request to acquire the resources. The communication mechanism should be abstracted in a fashion so that it can be easily express patterns such as a client requesting multiple resources, or model the conflicting access to a resource. At the same time, the abstraction should be general enough to allow for the easy addition of new clients or resources, while avoiding the specification of details about how the request mechanism is implemented. In a purely software implementation of system, the addition of a new client can be as simple as adding a function call. The operating system provides high level communication and synchronization primitives that can be used to implement the communication. In a hardware implementation, however, this new client will require the addition of extra computation capability as well as wires to handle the new data transfer path and to arbitrate between the clients. The addition of this new hardware in turn affects the communication path. As a result, any such addition requires extensive redesign, even

of parts of the design that are not directly affected. In this case, the addition of this new client will affect the hardware implementation of the second resource, which in turn will affect the design of other clients that may need to communicate with it.

Modern hardware designs, such as Network-on-chip and system-on-chip, take full advantage of the high abstraction level of computation to express complex systems. However, the lack of a corresponding clear high level abstraction for communication in digital hardware design presents a major challenge [Hemani et al., 2000] [Pestana et al., 2004] [Kumar et al., 2002] [Gerstlauer et al., 2005].

In this work, we address the problem of the lack of a useful communication abstraction for high level design of computer hardware. The proposed solution we present is a novel approach to high level computer system design. This methodology uses a communication and synchronization abstraction, that is as abstract as communication primitives used in software design. This abstraction is flexible enough to model a wide variety of different communication patterns. The solution we propose differs from previous approaches as it allows the designer to use a variety of features without restriction, such as multiparty communication and variable party communication. In addition, features like conjunction and disjunction of synchronization events and the composition of these primitives are fully supported. A novel communication scheduler enables the generation of structures that can greatly speed up simulation and allow for hardware synthesis of models described using this methodology. Further, a strong underlying formal model enables the creation of model checking and design verification tools.

1.2 Introduction to Synthesis

Circuits described using GTL, or at the gate level can be created manually for small systems, and until the late 1970's this was the norm. As the circuit complexity grows,

there arises the need for the move to RTL descriptions of circuits. A survey done in 1975 about the then emerging field of Register Transfer Level design [Barbacci, 1975] was one of the first to define the synthesis of a system as the conversion of a symbolic representation to a physical one. Gajski et. al. created a graphical aid to visualize the relationships between the domains of design [Gajski et al., 1992]. In terms of the domains in Figure 1·1, the arrows from the behavioral to the structural represent *synthesis*. The creation of a gate or transistor level structural description is called *physical synthesis*. *RTL synthesis* involves the generation of clocked registers and combinational circuits from a series of register transfers.

Finally, at the system level, *high level synthesis* involves the generation of complex structures from algorithm or software behavioral descriptions[Gajski et al., 1992]. While synthesis alone moves from one domain to another at the same level of abstraction, another important ability is to move from a higher abstraction level to a lower one in the same domain. This ability is referred to as *refinement*. In the structural domain, each subsequent level of abstraction is defined as a collection of elements from the lower one. For example, a gate is a collection of transistors, or a register and its update logic is a collection of gates. As a result, design refinement in the structural domain consists of replacing a higher abstraction level component with the group of lower abstraction level components that make it up. A final transformation is *optimization*, which refers to the improvement of a design in the same domain, within the same abstraction level utilizing some target characteristic. A common example of this would be the area and timing optimizations that are performed on designs.

Tools in the EDA field cover all three domains and primarily exist perform one of the three transformations : synthesis, refinement or optimization. As all circuits eventually need to be converted to a physical representation, we can conceive of

the design process as starting by defining the behavioral domain representation of a system at a high level of abstraction. Then, utilizing a set of tools, a structural description at the same level is generated. Finally, these descriptions are moved down the levels of abstraction until a physical domain transistor level circuit is obtained. This represents the *design flow*. The traditional design flow, especially for RTL circuits is seen here in Figure 1·3.

The idea of generating hardware from software-like descriptions was proposed in [Mead and Conway, 1980]. The steady increases in the level of abstraction were helpful in the creation of new tools and methodologies. First, tools for synthesis from each abstraction level were created and then used as the base components for the next abstraction level. As a result, valuable work done previously for tasks like area based optimization or physical place and route for layout is reused by methodologies that work at higher levels. Further, it meant that newly created tools only needed to establish a way to convert the input behavioral description into a structural description that could be reduced to the next lower level of abstraction.

The currently used synthesis methodology can be seen in Figure 1·3. The designer creates a behavioral description of a system in a RTL HDL such as Verilog or VHDL. By simulating the design, its behavior can be verified, leading to its iterative improvement. Once satisfied, the design is passed to the synthesis tool, which applies information about the technology library to create a netlist. This *netlist* is a gate level structural description of the circuit. It is passed through an optimization step to create an optimized netlist. Finally, the netlist is passed through the physical synthesis tools, which can create a complete layout from a basic gate layout, and then place and route wiring. The basic cell layout can come from the technology library or can be created by the user. Automated place and route tools use the user defined cells to generate a complete layout for the system and create wiring to connect these

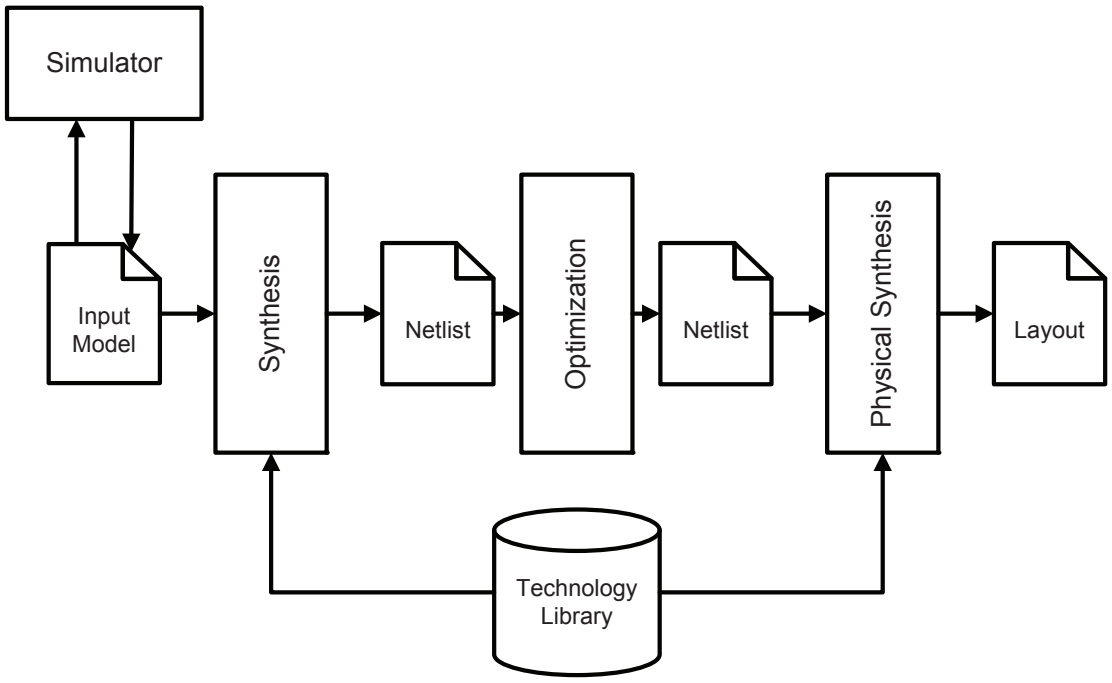

Figure 1·3: Traditional design flow for Register Transfer Level descriptions. The input is a register transfer level behavioral specification created in a HDL like Verilog or VHDL. The first step of the process generates a gate level structural netlist. After optimization, this netlist is used to generate a physical transistor level layout.

cells. This will create a functional physical layout for the whole model. A sequence of steps, where data is sequentially processed by a series of tools, is referred to as a *tool flow*. This presents a RTL synthesis tool flow.

The size of modern systems has been continuously rising, along with the design effort necessary. This makes the move of the behavioral description to a higher level than RTL extremely likely. In [Gupta and Brewer, 2008], Gupta et. al. demonstrate the financial benefit of high level design methodologies. They estimate that the design cost of an integrated circuit is US$15M with the use of tools at abstraction levels higher than RTL, as opposed to almost US$342M for a pure RTL approach. This is a 20x benefit to using such methods to create circuits, as opposed to the RTL abstraction based design. Modern high level design approaches have been gaining increasing acceptance [Martin and Smith, 2009] and are seen as the most important way to reduce design costs and design effort while simultaneously increasing the system size. In order for any high level design approach to be successful, there are many factors that must be considered, not the least of which is how they abstract communication and the mechanism they provide for the creation of lower abstraction level descriptions.

In Section 2.1, we present the current state of synthesis from high level descriptions and discuss the significant issues preventing their widespread use. In subsequent chapters, we discuss how the novel methodology presented in this dissertation does not suffer from the same drawbacks.

1.3 Motivation

Over the past few decades, successively more abstract models of digital hardware systems have been developed, allowing designers to encapsulate and contain their effective complexity. Beginning with the schematic transistor representations that were used by designers to replace the physical layout, there has been a move towards

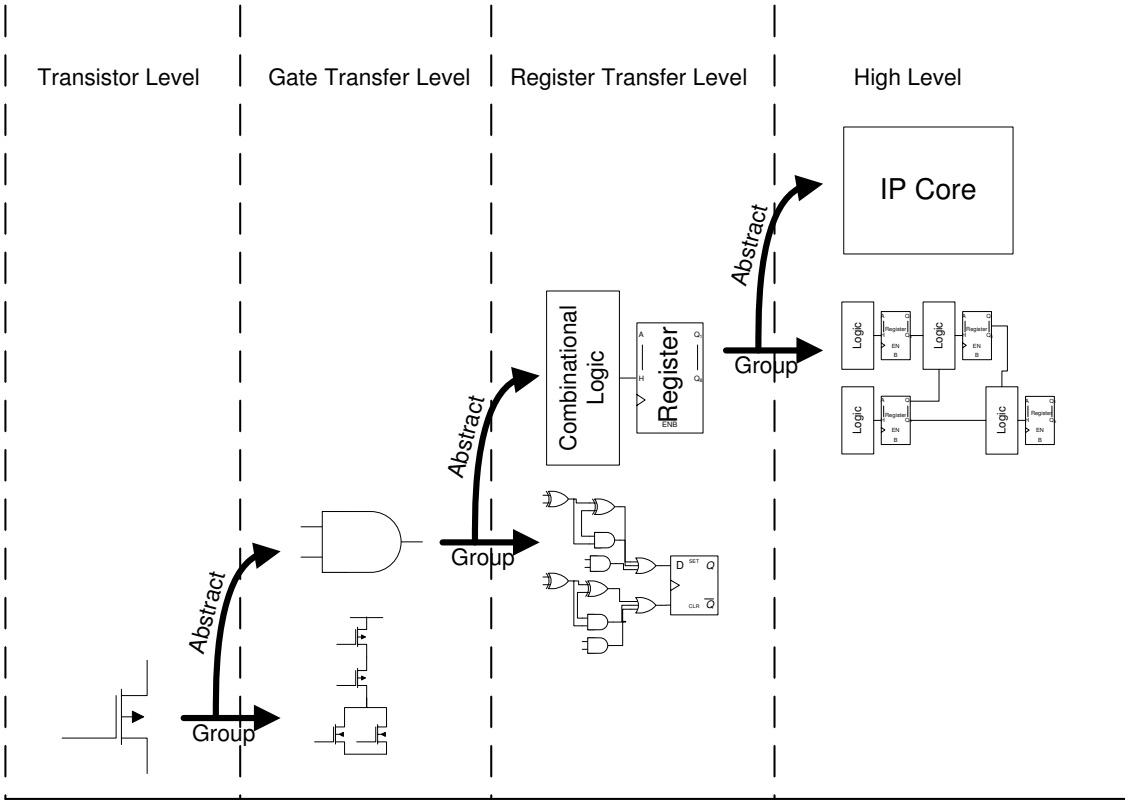

Figure 1·4: Levels of abstraction in the structural domain. The level of abstraction to represent computation has been rising, but that of communication has remained fixed.

expanding the gap between the physical and functional representations of a circuit, with each new level built upon the old. An increasing portion of modern design work is done at the system level, where designers are able to simply choose soft cores built by other vendors, that can perform complex computations, and integrate them into one system. The designer's task is to create and model the increasingly complex communication between vendor provided cores.

At the system level of abstraction, there is an open question of what design abstraction should be used and several high level models have been proposed for this purpose. The main focus of abstraction into a higher level model has been in one of two areas : computation and communication. Advances in compiler techniques and high level synthesis (HLS) techniques have provided strong support for high level description of computational tasks. High level modeling of communication and concurrency, on the other hand, has only seen success in special domains such as digital signal processing and synchronous programming. For broader applications, the recent trend of transaction level modeling (TLM)[Cai and Gajski, 2003], characterized by using software function calls for communication and synchronization, has become a prevailing standard. TLM is especially useful for bottom-up assembling of components of various natures into a common modeling framework such as SystemC [Automation, Design and Committee, Standards, 2006]. In software design, communication is performed dynamically, using support from the operating system. Software communication architectures, like message passing, or dynamically created sockets are not viable communication primitives in hardware. As software function calls are fundamentally only a language construct for code reuse and modularity, their use for modeling communication relies on the mechanisms of the underlying operating system, such as scheduling-related system calls and the thread library. In the case of SystemC, this involves the manipulation of the event calendar via the wait() and

notify() calls or similar methods. This approach is effective for functional simulation, but is not amenable to hardware synthesis or formal verification. The lack of a good high level way to represent different communication patterns is an increasing problem as designs become larger and communication plays a larger role in system design [Gerstlauer et al., 2005].

The flexibility of high level models in the abstract specification of systems is offset by the difficulty of synthesizing the hardware equivalent of such a description. In survey of the field, Sangiovanni-Vincentelli et. al. describe the "lack of clear unambiguous synthesis semantics" as a hurdle towards the adoption of existing high level methodologies such as SystemC [Sangiovanni-Vincentelli, 2003]. Traditional low level register transfer level (RTL) descriptions are easy to synthesize, but are too detailed to allow for reconfiguration to explore the design space and for functional verification. Thus, modern systems are typically maintained as two separate models. A high level model, in a language such as SystemC, that allows for functional and transactional behavior description and verification. Simultaneously, a low level RTL model must be maintained for behavioral synthesis. This duplication leads to a doubling of design time and effort and presents a semantic gap that is a source for errors and requires additional manual verification.

Recently in the EDA field, there is a renewed interest in using rendezvous for synthesizable system descriptions. A *rendezvous* is a classical communication mechanism for modeling communication. It appears in languages such as Handel-C [Celoxica Ltd, 2003], Haste [de Wit and Peeters, 2006], and most recently, SHIM [Edwards and Tardieu, 2005]. It provides a similar level of abstraction as software functions, but is based on formal foundations such as the Communication Sequential Processes (CSP) [Hoare, 1978] and the Calculus of Communication Systems (CCS) [Evangelist et al., 1989]. More interestingly, rendezvous naturally support multiparty

features such as disjunctive and conjunctive composition, which enable the modeling of choice and synchronization among multiple processes. However, due to the difficulty of implementing multiparty schedulers, contemporary rendezvous-based concurrent languages tend to leave out conjunctive composition or even both types of composition.

In this dissertation, we present a solution to the problem of a high level design methodology that is synthesizable. This work utilizes the idea of rendezvous as a powerful communication abstraction mechanism. This methodology, called *Lyra*, is presented as a series of communicating processes, which can synchronize via rendezvous. The abstract concept of rendezvous are presented concretely using two basic language primitives. The use of abstract rendezvous features are allowed so that designers can easily harness them. For example, rendezvous features such as disjunction are presented as simply multiple ways for the process to make progress. The use of the full range of rendezvous features presents the problem of nondeterminism. *Nondeterminism* refers to the case where there exist multiple methods for the system to make progress. This is an unavoidable feature in high level languages as it is a side effect of allowing partial specification of a design, an important tool in raising abstraction levels. As raising the abstraction level of communication has only now become a priority, existing approaches choose to resolve nondeterminism at the specification level. Typically, they achieve this by limiting the sources of nondeterminism, i.e. the features of rendezvous being used. Other approaches shift the burden for this resolution on the designer. In contrast, Lyra is capable of modeling complex communication patterns in a succinct fashion. The introduction of a novel *communication scheduler* allows this approach to be efficient for hardware synthesis while retaining the flexibility of communication primitives. The communication scheduler is able to resolve, at run time, the behavior of the system, and using some

optimization criteria, is able to ensure that the system functions in a deadlock free way. The generation of the communication scheduler is based on two major graph based tools. The first, called a *Transition Relation Graph*, is a represention of the relationships between different transition edges in the system and enables the efficient identification of the relationships that can create valid progress. The second, called the *Occurrence Relation Graph*, uses the relationship information to create a graphical representation of the communication complexity of the system and identify the communication problems that need to be resolved. The use of *policies* allow the separation of the heuristic used from the general approach, and present a way for the scheduling approach to resolve the same problem for different criteria. We also propose a *formal model* for Lyra, based on an extension of the Extended Finite State Machine formalism and on the definition of rendezvous. This formal basis is used to create the definition of the communication scheduler and clarifies the advantage of using a policy based approach.

The remainder of the dissertation is organized as follows. Section 2 discusses related work in high level synthesis as well existing rendezvous based design approaches. The solution to the problem of a high level language with a powerful communication abstraction is proposed in Chapter 3. An overview of the communication scheduling mechanism is developed in Chapter 4. The graph based tools for the construction of scheduler and policies for heuristic resolution are presented in Chapter 5. The underlying formal model for Lyra is developed in Chapter 6. A description of the tool flow that implements this methodology is covered in Chapter 7. A short examination of the communication complexity using the graph based tools developed for this approach is examined in Chapter 8. The use of the proposed methodology is demonstrated in Chapter 9 using the empirical example of a large system modeled at multiple abstraction levels. Chapter 10 develops some additional examples, and

demonstrates the benefits of using the communication scheduler. Finally, we present some conclusions and future avenues of work in Chapter 11.

Chapter 2

Related Work

In this chapter, we present an overview of existing digital hardware synthesis from high level behavioral descriptions. We examine some of the fundamental limitations with contemporary approaches for high level modeling of digital hardware systems. We describe the issue of nondeterminism, a fundamental problem in the high level abstraction of communication. We present the concept of rendezvous and describe previous work done on their usage. We describe a system developed for the classification of rendezvous based approaches.

2.1 High Level Synthesis

Synthesis is the conversion of a behavioral description into a structural one. This is usually implemented as a program that forms part of a tool flow, a set of programs that accept as input a higher abstraction level behavioral description and produce a lower abstraction level physical representation. High level modeling, where the input behavioral description is at a higher level of abstraction than RTL, is an increasingly important area [Sangiovanni-Vincentelli, 2003]. This form of design description is used mainly for the behavioral specification of entire systems. The increased flexibility at this abstraction level lends itself to modeling behavior that can range from software implementations of algorithms to collections of a large number of register transfer level models.

The implementation of the synthesis tools varies depending on the input language,

and the abstractions of computation and communication in use. While an exhaustive compilation of all proposed and existing high level design methodologies is beyond the scope of this work, most approaches, specially at abstraction levels higher than register transfer level share some common features.

In the remainder of this chapter, we examine the 3 different approaches to the design of a high level methodology. We discuss the fundamental differences between software programming approaches and hardware description languages in terms of their implicit assumptions. We also survey some existing techniques that allow for the limited synthesis of a software description into a RTL level structural hardware description. We examine the utility and drawbacks of nondeterminism in the context of high level description of synthesizable systems. We examine contemporary approaches to high level modeling based on C/C++ programming languages. Then we introduce the concept of rendezvous, a mechanism that performs communication and synchronization as one atomic action. We also examine existing rendezvous based hardware design approaches, in terms of the rendezvous features and modeling flexibility they allow.

2.2 High Level Synthesis Methodologies

There have been three basic approaches to the creation of high level design methodologies that are popular in industry today. Most of these have evolved from prior attempts to raise the level of abstraction past RTL for behavioral descriptions. While a thorough treatment of this subject can be found in [Gupta and Brewer, 2008] [Martin and Smith, 2009], they have been broadly classified into 3 categories seen in Table **??**

1. Software program based

2. Extension of RTL design

3. New methodologies

The first approach is to reuse the software based approach to model hardware. In other words, software programs are used to simulate the behavior of digital hardware. In some cases, these simulations can be synthesized into digital hardware as long as the underlying software model is restricted to a small subset of possible features. As hardware systems and software programs exhibit very different properties, arising from fundamental differences, the restriction of software features like threads, or dynamically allocated memory, which have no hardware analogues, is an essential part of modeling hardware using software. A good example would be a tool like Catapult C by Mentor Graphics [McCloud, 2004]. Catapult C is capable of producing a hardware implementable circuit from a C program, given some constraints on the features of ANSI C [ANSI, 1989] used, such as an absence of pointers, the lack of function recursion, and the use of only fixed width primitive data types.

A second popular approach has been to extend existing hardware design methodologies to a higher level of abstraction. A good example of this is Verilog[IEEE, 2001], a standard for RTL design, which was extended to SystemVerilog [Accellera Ltd, 2004] that brought to it features of high level methodologies. The primary advantage of this approach is that since the underlying model is meant to describe hardware, a subset of the higher level methodology remains provably synthesizeable. Additionally, if the syntax remains compatible, then the design work needed for the integration of existing designs into a new high level system description is reduced. The main pitfall of this approach is that the underlying model has severe restrictions due to its roots in hardware design. In SystemVerilog, [Accellera Ltd, 2004] this resulted in the partition of the methodology into a synthesizeable subset and a non-synthesizeable one. This reintroduces the problem of having to maintain separate behavioral and synthesizeable models.

The third approach is the design of a new methodology while using familiar language syntax. Typically, this means that the new methodology has an underlying set of assumptions that can model hardware, but the language that the approach uses is similar to, or shares elements in common with, software development languages. A simple example is HandelC [Celoxica Ltd, 2003], an approach that presents a language similar to C, but the underlying methodology communicates via hardware friendly primitives. The primary advantage of such approaches is that the lack of legacy restrictions on behavior allows such methodologies to describe behaviorally systems that have valid hardware descriptions. However, such approaches typically lack a formalism of the underlying communication model [Celoxica Ltd, 2003].

In the case of software approaches that are used to serve as high level hardware models[McCloud, 2004] [Cadence Inc, 2008] [Wakabayashi, 1999], the differences in the implict assumptions between software and hardware design methodologies become roadblocks. Software approaches inherently assume sequential execution, where one line is executed after the next. Concurrency in software programs, in the form of threads or processes, must be explicitly managed by the programmer. However, this concurrency is not completely controlled by the designer and depends on the operating system's scheduler and provided threading libraries. In contrast, hardware modeling methodologies assume concurrency, or provide a clear mechanism for it within the methodology. Software programs typically make heavy use of dynamically allocated data, and language features such as pointers and recursion which lack hardware analogues. Most importantly, software methodologies have communication methodologies that lack a description in the language but depend on OS semantics. To communicate between two software threads, the operating system plays a significant role. As a result, high level design methodologies that try to synthesize digital hardware systems from existing software programs, rely on the restriction of the input

language. The resulting language subset can be used to satisfy an internal hardware centric model and can be synthesized into hardware.

While there are many competing approaches that approach the problem in different ways, one of the main problems that any HLS approach must face is that of lack of formalism[Gajski et al., 1992]. Most high level approaches lack a formal underpinning, which in turn causes the lack of a clear direction for algorithms that convert the description into hardware. As a result, ad hoc mechanisms are employed to model important constructs that occur in hardware. Formal model developed after the methodology only express a subset of the features available[Man, 2005b] [Man, 2005a], making them less useful to designers.

In the next section, we identify the desireable features in a high level design methodology, and categorize them into properties inherent in the methodology and those that arise from the choice of the input description.

2.3 Properties for High Level Synthesis

While HLS methodologies are widely varied in their goals and implementations, there are some basic factors that we can say are necessary for any high level hardware design methodology. These are based upon criteria discussed in [Gajski et al., 1992] for the general methodology criteria and from [Edwards, 2005b] and [De Micheli, 1999] for the language criteria.

The properties can be divided into properties based on the methodology itself, and those that are dependent upon the language that is used for the methodology. This is an important distinction, but one that is sometimes useless in practice. In many cases, the methodology and the language that supports it are so closely tied together that the properties become interchangeable. This distinction, however, allows us to compare some very differing approaches to see their properties.

Methodology	Language Specific
Formal Theory	Concurrency
Flexible Modeling Semantics	Communication
Abstraction Levels	Data Type Support
Nondeterminism	

Table 2.1: High level design criteria

A good high level synthesis methodology should firstly posses some formal underpinning for the system so that models can be converted into hardware. In the case of many methodologies, especially those that try to use a software paradigm to model hardware,such as ANSI C [ANSI, 1989], it is their ad hoc nature that prevents their synthesis to hardware. The lack of a coherent formal model for high level methodologies means that there is no consistent methodology for the creation of hardware circuits. Furthermore, the formal model is essential to perform design verification and model checking. In modern systems, verifying the functionality of the system is an increasingly important part of their design. The use of formal verification tools allows for the design of larger systems. In many approaches for high level modeling, the final formal model incorporate a multitude of differing incompatible formal models, it becomes impossible to reconcile all the formalisms to create a provably correct design [Edwards et al., 2001]. This problem has been addressed previously in [Gajski et al., 1992] and [Camposano and Wolf, 1991].

The primitives that are proposed by the methodology must lend themselves to usage in a wide variety of modeling scenarios. As systems can be modeled in different ways, and different types of systems possess widely varying characteristics, it is important that the basics of the methodology can be adapted for use in all these cases. In the absence of such flexibility, the methodology designer is forced to create

a new primitive for each new scenario. This results in the creation of niche modeling methodologies that are capable of describing particular systems, but are not general. As high level models lack of a unifying formal model, this means that large designs that may require multiple different models will not be synthesizeable.

High level approaches generally possess the ability to model multiple abstraction levels, or at least use some design basis that supports design refinement[Gajski et al., 1992]. Refinement is the term used to describe the conversion of a more abstract model into a more concrete one, usually by the addition of constraints such as timing information, area or power guidelines. This is an important step for any design methodology, and any new high level methodology should be able to perform some level of refinement. Since high level tools maintain support for a wide variety of abstraction levels, their formal models are a mixture of behavioral, finite state machine, and register transfer formalisms, with no model for the entire approach. As a result, it becomes impossible to synthesize the full set of possible designs and features that the approach supports. Another important factor is that while high level approaches present abstractions for computation that can be easily refined, their abstraction for communication cannot always be. Further, the lack of a standard way to abstract communication means that different designers abstract communications in incompatible ways even while using the same high level approach.

Nondeterminism is a property of high level models that allows the designer to succintly express complex behaviors, by leaving unknown or undesired behavior unspecified [Armoni and Ben-Ari, 2009]. This reduces the size of the behavior that the designer must specify, reducing design time. However, nondeterminism is an undesireable property in a synthesized description. As a result, a good high level design methodology is capable of accepting nondeterminism in its input specification, but has a formal way to convert this specification into a deterministic one. This allows

the designer to flexibly express the system while seeing predictable results. The role of nondeterminism is addressed in depth in Section 2.5.

Another set of considerations come about from the language used to implement the methodology. The primary cause for these issues is the disconnect between software programming languages and hardware design languages.

2.4 Differences between Hardware Description Languages (HDLs) and software programming languages

As high level hardware descriptions can be sometimes expressed as algorithms and due to the larger number of designers familiar with software programing, many approaches try to make high level design as close to software design as possible [Sangiovanni-Vincentelli, 2003]. The basic differences between hardware design and software programming, many of which are implicit in the assumptions made prevent the easy use of a single approach that is capable of being both a software programming and a hardware description language.

The most important of the differences between software programming and hardware description languages, is the matter of concurrency and its description. In software programming languages, the code is assumed to be executed sequentially. Concurrency is explicitly implemented, either using threads [Liao et al., 1997] or by using keywords such as "par" in BachC [Kambe et al., 2001]. Hardware design languages, on the other hand, usually have concurrency defined as an implicit part of the language, such as in Verilog [IEEE, 2001] and VHDL [IEEE, 2000]. In addition, they support the notion of a "nonblocking" or parallel assignment. This fundamental difference in the definition of concurrency affects the design of the approach immensely.

Many high level approaches rely on the use of the hardware centric definition of concurrency, where each module or structural unit is assumed to be perform-

ing computations and communications in parallel[Gajski and Ramachandran, 1994]. Approaches that extend existing hardware design semantics, or that have created their own language are able to work around this easily. However, approaches that attempt to extend software programming to high level modeling and synthesis are forced to define new semantics. In the case of languages like SystemC [Automation, Design and Committee, Standards, 2006], each concurrent action is explicitly created and registered with the simulation engine. Thus, while the global notion of implicit concurrency is not present, the language semantics allow the explicit declaration of concurrent blocks of sequential code.

Communication as a language property that can be examined in all three contexts differently. In the case of hardware design languages, communication usually happens over fixed channels, but can support reactive inputs, through constructs like wires. In general, communication for RTL and lower HDLs is over wires that can be bidirectional and are of fixed size. Wires model hardware systems well and allow for bidirectional communication, synchronous communication and multiparty communication. In software programming languages, the fundamental method of communication is a function call, a unidirectional, asymmetric code reuse mechanism. Function calls can be used to model more complex communications by making the new modules for communication primitives. Function calls do have some advantages over the use of wires - their support for different data types and the flexibility of their composition. However, they have limited support for bidirectional communication, cannot handle multiparty communication well, and cannot model reactive inputs. New methodologies usually introduce their own communication primitives. Most methodologies allow for multiple communication primitives, to model different kinds of communication patterns.

The basic representation of data is another intrinsic property of the language

that affects the final approach. In the case of hardware design languages, data types are usually of fixed size, and do not support pointers. In software programming languages, data types can have variable sizes, or can be implemented as pointers. Most new approaches usually assume data types to be fixed in most cases, but allow for some flexibility to be able to model software patterns as well.

2.5 Nondeterminism

Nondeterminism in an important property in any high level synthesis approach. Nondeterminism refers to the case where given an input and a current state for a system, there are multiple possible next states. To put it in terms of the approach, nondeterminism is the property that arises from the ability to incompletely specify behavior of the model. This flexibility is crucial for high level approaches as it helps keep the input size small, while maximizing the behavior of the system. The presence of nondeterminism allows the modeling of complex practical designs in a simple fashion.

The concept of nondeterminism was first proposed in [Rabin and Scott, 1959]. In it, nondeterminism was proposed in the context of finite automata. A few years later, [Hopcroft et al., 1979] further formalized the concept of nondeterministic finite automata (NFAs). The equivalence of NFA and their deterministic counterparts was established, thus allowing the properties of deterministic finite automata to be applied to NFAs. In [Carrol and Long, 1989], a way to implement circuits from NFA descriptions is covered. As there was an equivalence between DFA and NFA, there is a way to reduce nondeterminism in a system. A clear formal proof of the power of nondeterminism to reduce the complexity of input descriptions is given in [Drusinsky and Harel, 1994]. Effectively, the presence of concurrency and nondeterminism in an input description exponentially reduces its size.

Nondeterminism is a practical mechanism for avoiding the complete description

of a system. In doing so, nondeterminism helps raise the level of abstraction. This was first pointed out in [Dijkstra, 1975]. By delaying the specification of implementation details, the presence of nondeterminism allows the designer to create the overall system behavior instead of the details of each stage.

A complete survey of nondeterminism is presented in [Armoni and Ben-Ari, 2009]. In it, the classification of nondeterminism based on 6 major properties is proposed and studied. [Armoni and Ben-Ari, 2009] also presents a distinction between nondeterminism arising from concurrency and nondeterminism arising from the property of an automata. A hardware implementation of a sequential process cannot contain nondeterminism as, at any given instant, the hardware will be in a fixed state. In order to ensure the creation of deterministic hardware that can be debugged, it becomes important that there exist a construction that can convert the nondeterministic system into a deterministic one in a fixed fashion. While the construction of NFA from a DFA is discussed in [Carrol and Long, 1989], the resulting deterministic automaton is exponentially larger.

The basic use of nondeterminism in high level approaches is clear. Nondeterminism is a powerful mechanism to reduce the complexity of designs, while allowing the designer to refrain from expressing unnecessary amounts of detail. This mechanism allows for the easy increase of the level of abstraction, an essential part of high level design. Formally, we can show that nondeterminism provides for exponential reductions in input design complexity, while still retaining a path to synthesis.

2.6 C/C++ Based approaches

In general, high level synthesis approaches can be divided into three major categories - approaches based on using and extending existing software design methodologies, those based on extending hardware design methods and those based on new method-

ologies that use a familiar syntax. A popular starting language is C. The proponents of this approach claim that familiarity with the language and that the benefit of being able to co-design both the hardware and the software of a system give it a significant advantage [Edwards, 2005a]. C, a language originally developed by Kernighan and Ritchie [Kernighan and Ritchie, 1988] was intended as a low level language, with a very tight correspondence to the assembly language. The resulting approach provided syntactic structures that simplified design tasks, but retained a correspondence to the microprocessor architecture. However, some of the choices made resulted in problems when it came to synthesizing general hardware from a C description. The introduction of data types that were not inherently fixed to hardware representations, and the introduction of pointers meant that a generic C program could not be guaranteed to create hardware. As a simple example, if we consider an array, a basic hardware type, is represented in C as a pointer, it becomes clear that unless the synthesis tool performs analysis of the pointer and its usage, it cannot determine how to size the resulting array. Thus, all C based approaches only accept a subset of C for synthesis. Some popular C based approaches include Cones [Stroud et al., 1988], HardwareC [Ku and De Micheli, 1990], SpecC [Gajski et al., 2000b] [Fujita and Nakamura, 2001] and BachC [Kambe et al., 2001].

Cones, introduced in 1988, was a very early approach to high level synthesis from C. By considering a very limited subset of C, and disallowing features like unbounded loops or pointers, cones could create gate level hardware from a C description. HardwareC was a new hardware modeling methodology that aimed to extend C like semantics to hardware. Despite its similarity and the naming, HardwareC is fundamentally different from C in that the basic language does not support pointers, assumes a global clock, and expresses concurrency through the use of modules. Thus, there is no advantage of designer familiarity with C. Additionally, general communication

Hardware Based	Software Based	New Methodology
Verilog	Cones	SpecC
VHDL	Cyber	HardwareC
SystemVerilog	C2Verilog	BachC
	Transmogrifier C	
	CatapultC	
	C-to-silicon	

Table 2.2: Approaches to HLS

patterns are difficult to express in HardwareC. For example, modeling the arbitration of resources cannot be done abstractly, as HardwareC has no support for it. SpecC is fundamentally a specification language, subsets of which are synthesizable. Like HardwareC, SpecC is a new methodology whose syntax looks very similar to ANSI C. By introducing a few more keywords for parallel composition of statements, and by introducing separate channels for communication SpecC attempts to simplify hardware design. However, the lack of a clear definition for atomicity, and the absence of a way to define mutual exclusion presents some shortcomings. BachC was an attempt to bring the communication semantics of Occam [Barret, 1992] to C. As a result, BachC faces a similar set of challenges as Occam, and is explained further in depth in the next section.

Another direction that was considered was the use of more complex synthesis tools with C. The modifications made to the C design itself were very minor, i.e. either used a restricted subset of the input language or provided some bounds on software constructions such as pointers. Cyber, an approach introduced by NEC [Wakabayashi, 1999] [Wakabayashi, 2004], used a variant of C that prohibited recursion and pointers. C2Verilog was a synthesis tool that accepted almost a complete

set of ANSI C to create a Verilog circuit [Soderman and Panchul, 1998]. Transmogrifier C [Galloway et al., 1995] was a synthesis tool that used a small subset of C to generate hardware descriptions that could be easily used on an FPGA. The scenic design environment [Liao et al., 1997] was an approach that simplified the expression of concurrency by modeling each module as a parallel lightweight thread. Cosyma [Ernst et al., 1996], took a different stance and used the C based description to create a partition of the hardware and software environments. It would synthesize the subset of C it could, and allow the remainder to be executed as software. Additionally, attempts such as SpC [Semeria and De Micheli, 1998] tried to show that in many cases, pointers could be resolved fully at synthesis time, and therefore allow a commonly used C language construct in the synthesizable subset.

The newest of the C/C++ based HLS methodologies are CatapultC by Mentor [McCloud, 2004] and C-to-Silicon by Cadence [Cadence Inc, 2008]. Both of these are capable of using a very large subset of C and can create gate level netlists from the C model.

Finally, the most popular high level modeling methodology is based on SystemC [Automation, Design and Committee, Standards, 2006]. SystemC has wide support in the EDA industry for the specification of models at a high level. Communication in SystemC is handled by abstract channels. These channels are C++ classes implemented using wait() and notify() function calls. While the description is flexible from a software programming point of view, it is not possible to synthesize an arbitrary channel as it lacks a hardware equivalent. Further, the channels follow delayed update semantics, and behave more like registers in RTL HDLs like Verilog, rather than as wires or combinational paths. The lack of wires makes the expression of reactive systems in SystemC slightly cumbersome. Concurrency in SystemC is handled explicitly, where each module is declared as an independent sc_thread or sc_method.

SystemC sc_threads closely resemble software threads, implementing a form of co-operative multithreading and share many of the same properties. As a result, HLS tools like C-to-silicon and CatapultC cannot synthesize modules containing threads. SystemC methods are typically used to implement combinational elements. While clocks can be declared manually, the notion of clocking is implicit in SystemC, the execution of these models happens using a delta time update approach. Combined with the delayed update semantics of the communication channels, every communicating module in SystemC effectively behaves as a register transfer stage, containing the combinational update logic and the buffered data output. While SystemC supports the intrinsic C++ data types, synthesis tools usually prefer the use of SystemC specific types. These types support fixed width integers, fixed and floating point numbers and strings. Again, not all of these are synthesizable and depend heavily on the synthesis tool.

The style of modeling most used with SystemC to create the high level approach is referred to as Transaction Level Modeling (TLM) [Cai and Gajski, 2003]. While TLM is supposed to be a generic high level design approach, the reference implementation is in SystemC[Maillet-Contoz and Ghenassia, 2005]. As a result, features of SystemC came about due to the TLM semantics, and vice versa. The primary goal of TLM is to further subdivide the levels of abstraction and provide a path from an abstract, untimed, specification to a timed, cycle accurate computation based model. Some example TLM flows are presented in [Donlin, 2004], and show the heavy reliance of TLM models on the communicating process abstraction level of TLM. However, there are no formal models, or well defined synthesis flows for TLM. Additionally, portions of the specification are implemented using event function calls that modify the scheduler's event queue, making it difficult to synthesize directly. Finally, the TLM approach relies very heavily on the user, who needs to guide each step in

the refinement. As a result, the designer must be familiar, not only with software programming, but also, low level hardware design.

While the SystemC methodology itself lacks formal semantics and is ad hoc, there has been work on a subset of SystemC that can be described formally [Man, 2005b] [Man, 2005a]. Further, the basic form of communication, channels, are not close analogues of hardware wires. More importantly, SystemC is able to provide dataflow semantics, but the control flow semantics and synchronization are not handled in a scalable hardware friendly way. Finally, SystemC is not generally synthesizable, and leads to the requirement of maintaining a RTL implementation that can actually be implemented and a SystemC implementation that is used to verify the specification, leading to duplication of work and the potential for errors.

2.7 Introduction to Rendezvous

The concept of rendezvous was first introduced in CSP [Hoare, 1978] and CCS [Evangelist et al., 1989]. A rendezvous is generally regarded as an atomic, synchronous communication mechanism among processes. In its simplest bi-party form, two processes wait for each other at their respective rendezvous points where they expect to communicate. When both parties reached the rendezvous points, the rendezvous occurs, allowing both processes to continue to progress. In its more general form, as pointed out by [Joung and Smolka, 1996], we can define a rendezvous as an n party interaction with k roles. A role is a particular type of participant for the interaction and could be satisfied by a fixed or a variable set of parties.

We can then further classify rendezvous based on which primitive composition rules are allowed. *Conjunction* is the case where multiple rendezvous must occur simultaneously. *Disjunction* is the case where only one of a set of rendezvous occurs at a given time. A system can then be classified based on the allowed freedom of

composition of these basic rules. In summary, the use of rendezvous can be classified based on the following characteristics.

1. Multipartiness - the number of parties involved in each occurrence of a rendezvous

2. Variability - whether participants for each party of a rendezvous must be a fixed set of processes or a variable set

3. Composition Rules - the number of restrictions placed on the combination of the primitives below.

   (a) Disjunction - whether a rendezvous can be used in disjunction so that one of several choices should occur

   (b) Conjunction - whether a rendezvous can be used in conjunction so that several rendezvous must jointly occur

Figure 2·1 illustrates these features. It shows an example network of sequential processes in the form of state diagrams. *C1* and *C2* represent computing clients. M1 and M2 represent resource arbiters. The state transition edges are labeled with rendezvous. A transition along an edge requires the occurrence of all the rendezvous on the edge. With respect to the four characteristics of rendezvous:

1. The req's and rel's are bi-party rendezvous. As a convention throughout this work, we name the two roles "+" and "-". Exactly one "+" party and one "-" party should participate in each occurrence of a bi-party rendezvous. A multi-party rendezvous reset brings all processes back to their initial states from their respective current states. It has four roles, one for each of the four processes involved.

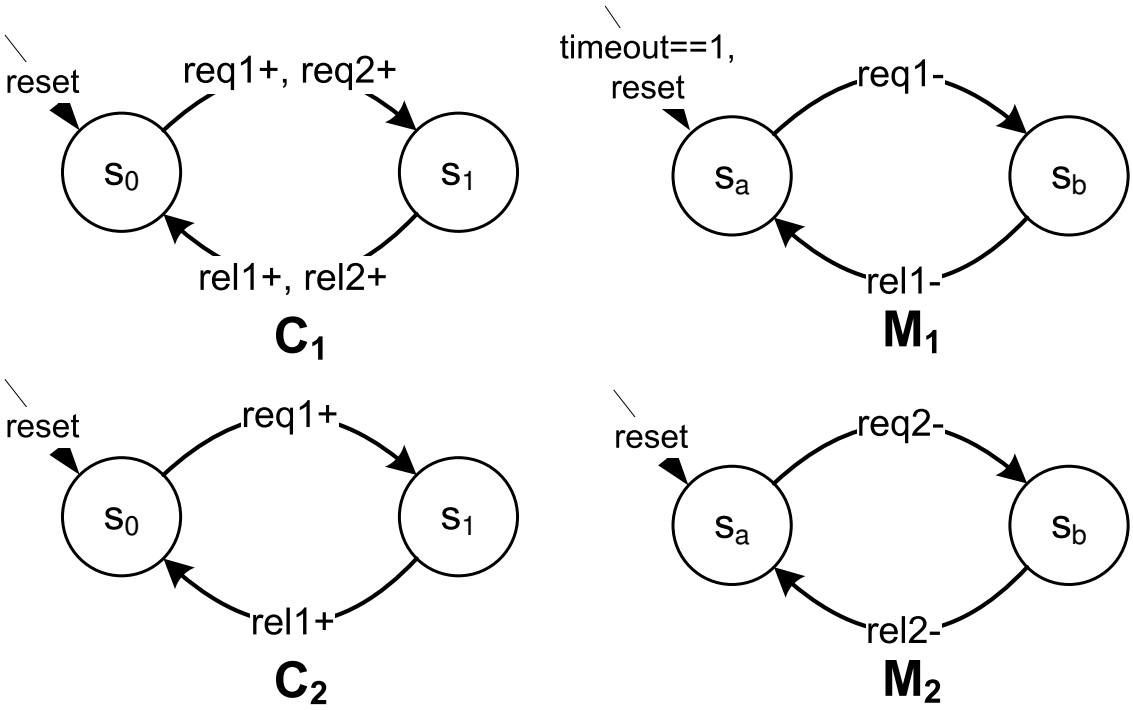

Figure 2·1: An Example Process Network

2. Req1 and rel1 are variable as their "+" roles can be assumed by either C1 or C2.

3. Every state has disjunction involved. For example, state S0 of C2 can transition to either S1 when req1 occurs, or to itself when reset occurs.

4. Process C1 requires the simultaneous occurrences of req1 and req2 to advance to S1. It will also activate rel1 and rel2 simultaneously to return to S0. Such simultaneous occurrence means conjunction.

As we can see from the use of conjunction and disjunction, in this case we can have both forms of composition from each state.

Theoretically speaking, multipartiness and variability can be synthesized from conjunction and disjunction. A multiparty rendezvous involving n-parties may be represented as (n-1) bi-party rendezvous used in conjunction. For example, the reset

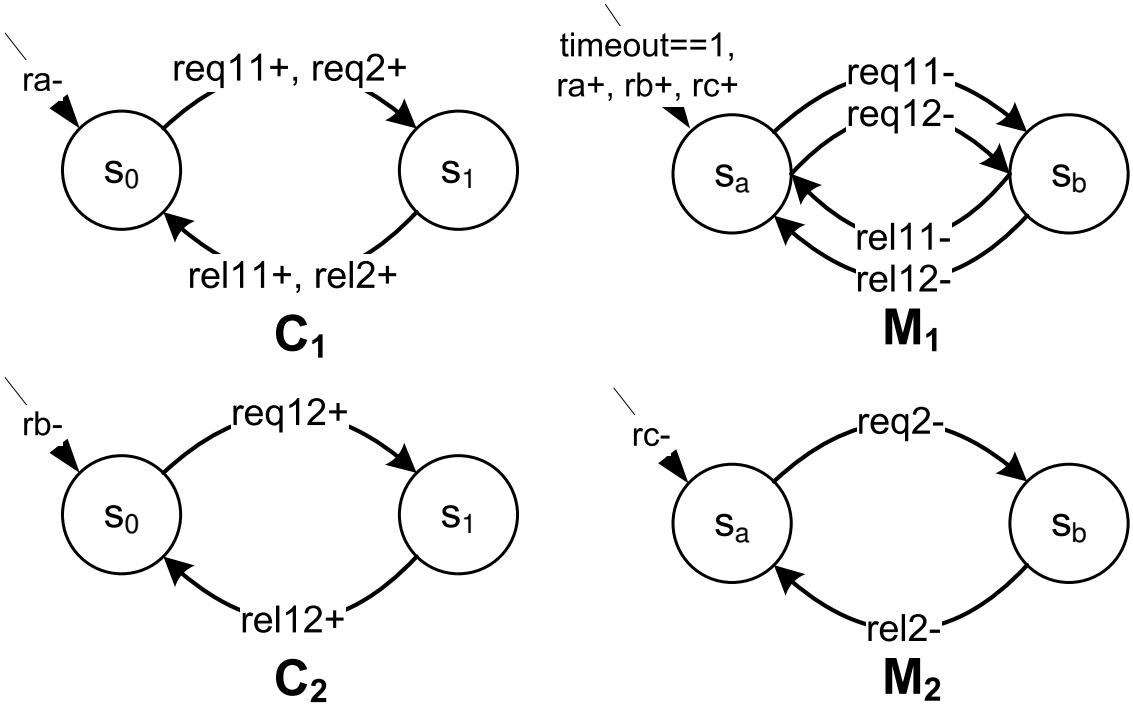

Figure 2·2: Decomposition of Variability

in Figure 2·1 can be decomposed into three bi-party rendezvous, say ra, rb, and rc. The reset label in M1 can thus be replaced by a conjunction of ra+, rb+, and rc+. The reset labels in C1, C2 and M2 can be replaced by ra-, rb-, and rc-, respectively. The result network has identical behavior as the original one. Similarly, a variable bi-party rendezvous involving n processes in one role and m in the other can be decomposed into n*m bi-party rendezvous. For example, the req1 in Figure 2·2 can be decomposed into 2*1=2 fixed bi-party rendezvous: a req11 between C1 and M1, and a req12 between C2 and M1. The same can be done for rel1. Figure 3 shows the resulting network after the decomposition of reset, req1 and rel1. More generally, a variable k-party rendezvous with ni processes for each role can be decomposed into ni bi-party rendezvous. The inclusion of multipartiness and variability make modeling convenient as they greatly reduce the number of symbols in the network.

Another concern addressed in [Joung and Smolka, 1996] is the composition rules

for conjunction and disjunction. When used in isolation, conjunction and disjunction are of limited value. However, their full and free combination, in the so called Discrete Normal Form (DNF), allows for the maximum expressivity of different communication patterns. However, a problem with this unfettered composition is that, while examining the possibility of occurrence of a rendezvous solely from the local context of one process, it becomes difficult to determine how other rendezvous may be composed at the remote party. These remote compositions may cause other rendezvous to be dependent on this one for occurrence.

2.8 Nondeterminism using Rendezvous

Nondeterminism, in the case of rendezvous models, can be simply defined as the case where, for the same input global state, there are multiple possible next states, each of which can be atomically reached through the occurrence of a different set of state transition edges. From a modeling perspective, high level models often use non determinism to abstract low level details. For example, when all processes are at their respective left states in Figure 2·1, either a transition of C1 accompanied by the occurrence of req1 and req2, or a transition of C2 accompanied by the occurrence of req1, is a valid step.

When a model is converted to an actual implementation, this nondeterminism needs to be removed. This can be achieved by either modifying the model through the addition of semantic constraints, or by imposing a run time scheduling policy for the resolution of nondeterminism. This problem can be solved at every run time iteration either in software by a scheduling core, or, in hardware, through a synthesized logic circuit consisting of arbiters and coordinators. As is clear, run time resolution of nondeterminism is not a trivial task. As this becomes the main task for each run time iteration, the communication complexity of the end system can be defined in

terms of the amount of nondeterminism present.

There are many sources of nondeterminism in rendezvous based models. Variability implies competition between different processes for the same role of a rendezvous. Disjunction, on the other hand, implies the competition amongst several transition choices. These two sources of nondeterminism respectively correspond to horizontal and vertical nondeterminism in [Joung and Smolka, 1996]. In both cases, the end result remains that the global state has multiple possible transitions, each of which result in different next states. The free composition of these properties further exacerbates the degree of nondeterminism. The simultaneous appearance of disjunction and variability multiply the degree of nondeterminism. Furthermore, conjunction can make an otherwise deterministic edge participate in a nondeterministic choice.

In terms of [Armoni and Ben-Ari, 2009], the nondeterminism in this approach is due solely to the choice of execution of the system based on the rendezvous. Even though the methodologies are inherently concurrent, and hence potentially another source of nondeterminism, since they are used for hardware modeling, the concurrent nature is assumed to be simply resolved by the creation of parallel hardware. An important note though is that unless specified, the approaches do not have an implicit method of resolution of concurrency that may arise from this parallel definition, and thus though the system can be created, when it executes it might become incapable of making progress.

2.9 Rendezvous Based Approaches

Communicating Sequential Processes, introduced by Hoare in [Hoare, 1978], was among the first process calculi to model a system as a network of sequential processes communicating over rendezvous. CSP, in its basic form, supports only disjunctive composition of rendezvous, and the use of multiparty ren-

Language	Multiparty	Variability	Disjunction	Conjunction
SHIM	X			
Ada		X		
Occam		X		
Handel-C		X		
Haste	X	X	X	X

Table 2.3: Feature Comparison

dezvous. Following the creation of CSP and CCS, many rendezvous-based concurrent languages were developed. These include Ada [Taft and Duff, 1997], Occam [Barret, 1992], Handel-C [Celoxica Ltd, 2003], Haste [de Wit and Peeters, 2006], and SHIM [Edwards and Tardieu, 2005]. Among these, Occam, Handel-C and Haste are all based on CSP, though their support for some aspects of rendezvous differs. A well-known language in verification community is Lotos [Bolognesi and Brinksma, 1987], which is based on both extended CCS and CSP. However, since it is designed as a specification language but not a programming/modeling language, we omit it in the comparison. Table 2.3 compares the rendezvous features that the languages support. The multiparty column refers to fixed multiparty barrier. The variability column refers to bi-party rendezvous.

The languages vary in their support of rendezvous features, and consequently in their degree of nondeterminism. The extreme case is SHIM, which is completely deterministic. The resulting model is a variation of a Kahn Process Network [Kahn, 1974] and provably possesses similar properties of determinism. It is free of variability and disjunction, and also forbids simultaneous write access to a shared variable. Scheduling of such a system is trivial and can be fully determined by the processes themselves.

Ada defines its rendezvous as a non-atomic communication between a "caller"

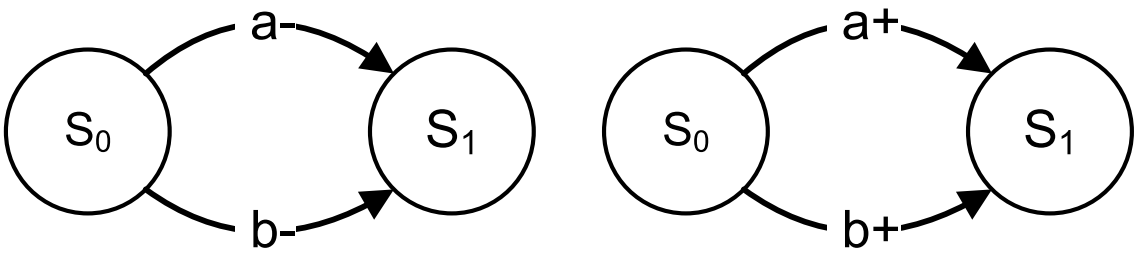

Figure 2·3: Unresolved Nondeterminism

and a "callee". It supports limited variability on the "caller" side. Ada proactively removes the nondeterminism associated to its variability by using an implicit queue in the "callee". The queue ensures that the callers are served on a first-come-first-serve basis. In other words, Ada uses time-based arbitration policy to prevent its nondeterminism. Occam strictly uses bi-party rendezvous channels. A channel connects exactly one reader process and one writer process. It allows disjunctive composition on the reader's side. So a process may wait to read from several rendezvous channels using the "Alt" statement. An Occam process may wait to read from several rendezvous channels, but cannot wait to write to several channels. The writer process must be committed to join the rendezvous channel it writes to. Thus, only the read side may have nondeterminism arising from disjunction. Fortunately, such nondeterminism can be resolved locally within the process by any fixed priority scheme that the process chooses. Therefore there is no need for global scheduling in Occam.

In Haste, only limited arbitration is done for variable rendezvous. In general, for a rendezvous to successfully occur between two Haste processes, at least one of them must be committed to joining the rendezvous. For the example of Figure 2·3, the two processes are not committed to either rendezvous due to disjunctive composition at their left states. Thus neither rendezvous can occur, resulting in a deadlock. Moreover, Haste does not support instantaneous data flow across conjunctively composed rendezvous. Therefore, it is not capable of modeling the example in Figure 2·3.

Handel-C prohibits variability but allows disjunctive composition. Its handling of disjunction is very similar to Haste and therefore suffers from the same drawback as pointed out in Figure 2·3.

In the case of the example in Figure 2·1, SHIM, Ada, Handel-C and Occam would be unable to model the conjunction of the rendezvous req1 and req2 in process C1 due to their lack of support for the conjunction primitive. The incomplete support for resolution of nondeterminism in disjunction for languages that do support it, makes Haste and Handel-C unable to model the example shown in Figure 2·3. Finally, all presented languages lack the support for combinational data flow across conjunctively composed rendezvous.

In summary, the existing languages support different sets of features of rendezvous and thus different degrees of nondeterminism. In Haste and Ada, local arbitration is performed to resolve nondeterminism introduced due to variability. But none of the surveyed languages performs the scheduling of global rendezvous.

2.10 Other High Level Approaches

While this work focuses on other rendezvous based languages, they are by no means the only forms of high level modeling. A recent high level language that is worth mentioning is Bluespec [Nikhil, 2008], despite the fact that it does not use rendezvous. A Bluespec model contains a set of atomic rules guarded by Boolean conditions. For communication between processes, Bluespec contains methods which can be asymmetrically invoked. A Bluespec method maps to a combinational circuit, which can perform some computation and update state. Bluespec supports implicit concurrency, but does not support nondeterministic model definitions, as the atomic condition guards are mutually exclusive. As we cannot place it in the framework developed in [Joung and Smolka, 1996] , we will instead analyze it using the methods developed

in Section 5.

Synchronous languages, a family of design languages based on Esterel [Berry and Gonthier, 1992], are in use to describe reactive systems. Esterel has some theoretical underpinnings in CSP, but eschews the use of rendezvous, instead replacing it with the notion of synchrony. In Esterel, synchrony basically refers to the assumption that all computations, updates, communications in the system happen atomically, and instantly. Esterel, thus implicitly contains the notion of instantaneous data flow. However, Esterel has no support for multipartiness or variability of communication, due to its deterministic nature.

High level design and synthesis approaches are being used to model systems at the behavioral level [Sangiovanni-Vincentelli, 2003]. Even though the level of abstraction of computation has increased, the abstraction level of communication has not. In transaction level modeling[Cai and Gajski, 2003], the modeling of communication is a significant factor in the complexity of the design. Extending software based models to perform hardware design is not feasible as it cannot synthesize a subset of the design[Automation, Design and Committee, Standards, 2006]. Furthermore, the lack of a formal model for the systems increses the complexity of generation and verification of arbitrary hardware designs[Man, 2005b]. The design approaches for RTL level hardware design, such as Verilog[IEEE, 2001] and VHDL[IEEE, 2000], provide only wires as primitives for communication. The lack of other primitives for the composition of communication and expression of synchronization makes them infeasible for use in behavioral level description. In this dissertation, we describe on a novel methodology that uses the concept of rendezvous for the abstraction of communication and synchronization. Unlike previous rendezvous based approaches, this approach allows for the full flexibility of composition of rendezvous[Joung and Smolka, 1996]. Using a novel communication scheduler, which is implemented as a software scheduler and

can be synthesized into a hardware module, this approach can express a wide variety of communication patterns, while retaining the ability to synthesize hardware.

Chapter 3

Lyra: An Introduction

In this chapter, we describe Lyra, a novel hardware design methodology that uses a rendezvous based communication abstraction. We describe the basic features of the methodology - the model of computation and communication. We briefly describe the communication and synchronization primitives available and how they are modeled. We finally deal with the need for nondeterminism in the description and the resulting need for a communication scheduler.

3.1 An Overview

For a modeling methodology, the selection of features of rendezvous that must be supported depends on its application. As we are targeting hardware design, it is important that common communication patterns in hardware be easily modeled. Combinational data flow is such a pattern, necessitating support for conjunctive composition. Further, conjunction is essential to model the all or none semantics that frequently appear in hardware. Disjunction represents a very basic primitive for choice and therefore must be part of any modeling methodology. As multiparty synchronization is very easy and commonplace, there should be a simple way for the designer to express it. Similarly, as variability allows a designer to easily express shared resource contention, it should also be explicitly supported. Thus, the choices of supported rendezvous features arise from practical concerns in the application of this methodology. The languages discussed were mostly aimed at software implemen-

tations, in which primitives like conjunction and multiparty communication are hard to implement. As a result, the previously discussed languages only support a small subset of rendezvous features.

The primary focus is in allowing the maximum expressive freedom in designing systems while at the same time keeping in mind that the end result must be synthesizable to hardware. The proposed modeling methodology shares features, such as support for multipartiness, variability and free composition of conjunction and disjunction primitives, in common with Haste. However, the use of scheduling and the flexible dataflow make it more expressive than Haste. Of the modeling languages discussed so far, the proposed framework is the most flexible in terms of allowing variability, conjunction, disjunction and composition of both, as well as supporting multi party rendezvous.

3.2 Description of Lyra

Lyra, the proposed approach uses rendezvous as the primary means of communication between networks of sequential processes. The two fundamental communication primitives are a variable participant, bi-party rendezvous and a fixed participant, multi-party barrier. The choice of these two types of rendezvous aims to meet the common communication patterns in hardware systems. The bi-party rendezvous typically models bi-party synchronization, or resource transactions involving choice and competition. Barrier models group synchronization among several processes. It is often used to represent a clock or a reset signal. As shown previously, a barrier is not essential and can be created using conjunction. However, as it represents a common pattern used in hardware design, it is included in the methodology. Note we choose to omit variable multi-party rendezvous since it is rarely useful in practice. If such a rendezvous is ever desired, it can be implemented as a set of multiparty rendezvous.

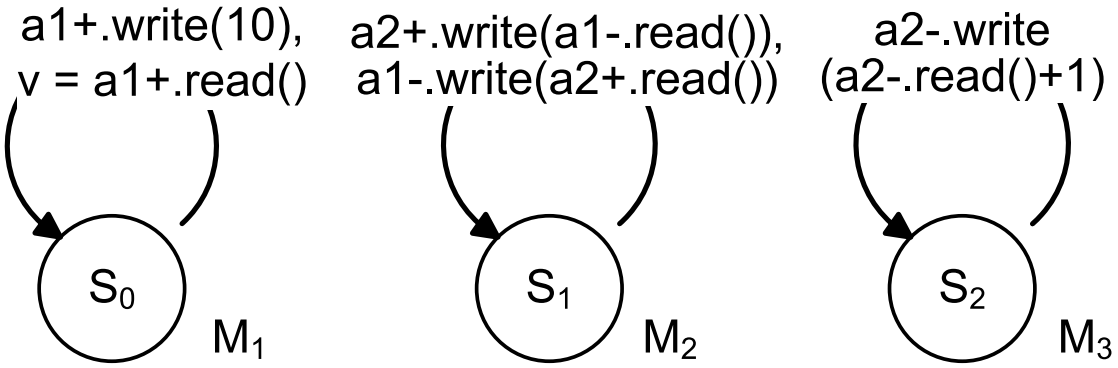

Figure 3·1: Synchronous data flow example

Processes are expressed as state diagrams, where the transition edges (TEs) are annotated with the rendezvous they are participating in, as well as the associated semantic actions and the conditions guarding the transition. For the process to transition along a given TE, all rendezvous on the edge must occur and the Boolean expression on the edge must be true. When a transition takes place, all statements on the edge are evaluated and updated atomically. For maximum expressive power, we allow both disjunctive and conjunctive composition of rendezvous. For data flow modeling, we permit the following:

1. Bi-party rendezvous can carry bidirectional data flow.

2. Multi-party barrier can have one writer of data and multiple readers.

3. Data flows synchronously between the input and the output of a bi-party rendezvous, and across conjunctively composed rendezvous. In other words, the input data of one rendezvous may be forwarded to another rendezvous instantaneously. This is similar to the synchronous data flow of Esterel.

The synchronous data flow is illustrated by the example in Figure 3·1. Rendezvous a1 forward a value of 10 to process M2, which immediately forwards it to M3 via a2.

M3 increments the value, and send the result instantaneously back to M2, which immediately sends it back to M1. Such synchronous data forwarding is useful to model combinational paths across hardware components. It enables similar expressiveness as synchronous languages such as Esterel for processes connected via conjunctive rendezvous. Synchronous data flow may lead to combinational cycles, which we consider invalid in our approach.

In the classic CSP theory, communication is solely via rendezvous. In practical implementations, however, this is not convenient. In the proposed model, in addition to rendezvous based communication, processes can also communicate via shared variables. The use of shared variables in our approach is partially motivated by the need for model reactivity. For example, when a process is busy doing computation, it may not react to queries from the environment. To realize reactivity, a separate process that has access to its state can be used to respond to the environment. In theory, it is possible to merge the two processes into one process and thus to keep the state private. But such an approach is against the general engineering practice of separation of concerns, and invariably results in huge state diagrams. After all, shared variable is common in multithreaded programs. Thanks to the atomicity assumption of state transitions in our approach, it is easy to resolve race conditions associated with shared variables.

To simplify the modeling of large dataflow circuits, the proposed methodology supports the use of combinational datapaths, as well as combinational signals. These can be used to provide a purely synchronous dataflow like Lustre [Halbwachs et al., 1991]. The combinational datapaths can be local to each module. The signals behave like Verilog's wires and can read from the datapath or registers, but cannot update a register. This is to ensure that all race conditions arising from register writes exist only inside processes, on atomic state transitions.

Lyra, the proposed modeling methodology, provides several benefits compared to existing rendezvous based languages, due to its support of multipartiness, variability and free composition of conjunction and disjunction. Its support for instantaneous synchronous data flow allows it to use model patterns similar to Esterel. Unlike Bluespec, where rule composition is hard, Lyra makes it easy for designers to specify the relationships between rendezvous. Compared to SystemC, the theoretical basis of Lyra makes it easier to generate formal semantics for models. Further, Lyra retains a clear path to synthesis of hardware.

However, these advantages come at the cost of the need of global scheduling. The use of these features of rendezvous introduces nondeterminism in the system that cannot always be resolved using local knowledge. Thus, in order to verify if a particular Transition Edge (TE) can occur, a Lyra model must check not only other roles of a rendezvous, but also all rendezvous that are related to it through composition, potentially involving the entire system. In the absence of a global scheduler, there exists the possibility of scheduling deadlocks. For example, in the system in Figure 2·1, if all processes are in their left state, then without a global scheduler, the rendezvous req1 is contested. Using only local knowledge, processes C1 and C2 might independently come to the conclusion that req1 should occur. When the processes attempt to make progress on this basis, a choice cannot be made on how req1 should occur, and as a result, the system deadlocks.

3.3 The need for nondeterminism in Lyra

The need for nondeterminism for high level modeling is two fold. Primarily, nondeterminism hides the details of implementation of communication by handing off the complexity of the problem to the tool rather than the designer. As a result, the designer is able to express complex communication patterns in a simple fashion, rather

than having to design a communication protocol. The second reason for the need for nondeterminism is that when this degree of freedom is allowed in communicating process based methodologies, nondeterminism helps reduce the level of complexity of the design, and manages one of the main problems with the use of finite state machine based approaches in modeling practical systems - i.e. state explosion. While the effects of state explosion are explored more in depth in chapter 6, the can be informally explained as follows: When multiple state machines are communicating, the resulting overall system, can in turn be expressed as a larger state machine. While the state space for each component state machine is added to the overall state machine's input state space, the overall output function is the Cartesian product of the overall input and the overall state spaces. As a result, the linear increase is amplified, and causes an exponential increase in the size of the system description. The value of nondeterminism is in providing an exponential decrease in complexity [Edwards et al., 2001]. Very specifically, the presence of conjunctive and disjunctive compositions in a system with inherent concurrency exponentially reduces the size of the system description [Drusinsky and Harel, 1994].

3.4 Communication Scheduling

The expressive power of the model, represented by the freedom of composition of disjunction and conjunction, as well as the inherent support for nondeterminism necessitates some external method to coordinate the communications between processes. To solve the problems due to nondeterminism, Lyra implements a synthesis friendly scheduler. A more detailed description of the scheduling framework is presented in Chapter 4. Additionally, the formal model for the scheduler is discussed in Chapter 6. The impact of the scheduler is further discussed in the Lyra tool flow chapter. In brief, the input Lyra system is analyzed to create a final graphical representation that

captures the basic sources of nondeterminism in the system. This nondeterminism is then resolved using a user defined policy, a pair of which are described in Chapter 4. By partitioning the problem into a static analysis phase and a run time scheduling phase, the resulting system is amenable to hardware synthesis, while retaining the run time decision making ability of a software scheduler. Thus, the scheduler forms the basis for both the hardware synthesis and the software simulation tools, which are further described in Chapter 7.

Chapter 4

Scheduling

In this chapter, we describe the novel communication scheduling mechanism that is used for Lyra. We examine the constraints and properties that a communication scheduler should posses. We then discuss scheduling policies that a scheduler can use, and their impact on the communication patterns in the system

4.1 Overview of Scheduling

The expressive power of the model, represented by the freedom of composition of disjunction and conjunction, as well as the inherent support for nondeterminism necessitates some external scheduler which can coordinate the communications between processes. Such a scheduler, based on a given scheme, can provide us with the best set of transition edges (TEs) to be simultaneously activated for the system to progress. The definition of best set of TEs is done on the basis of some predefined scheduling policy that allows us to choose between multiple sets of TEs. In the most general case, the proposed modeling methodology needs a scheduler to progress. Other languages circumvent the need for a central scheduler by limiting the expressive power of the system, by restricting the features of rendezvous. Another approach, used by languages like Haste, is to relax the assumption that the system is free of scheduling deadlocks.

The remainder of this section discusses the problem of scheduling and the properties of a scheduler. Section 5 presents the graphical methods, based on finding

relationships between state transition edges in the TRG and between minimal candidate schedules in the ORG. The graphical methods deal with static analysis, i.e. the analysis we can perform on a model, given no run time information. We discuss considerations that go into the design of a scheduler and introduce two possible choices for the scheduler policy. The analysis of the effects of these properties and the choice of policies will be in Section 4.3 after we introduce the graphical notation which we will be using for scheduler analysis. Finally, we will deal with dynamic scheduling, or the selection of a schedule given a global state. Dynamic scheduling is the scheduling we perform per run-time iteration of the system. Some of this work was previously discussed in [Venkataraman et al., 2009b].

4.2 Properties of Scheduling

In general, the scheduler is a process which receives the set of process and the global state as an input, and produces a set of TEs, which upon simultaneous occurrence, result in the system transitioning to a new global state. Additionally, the scheduler may contain within it a specific policy, which allows us to select between possible sets.

Given the proposed modeling methodology, we can formulate some basic principles that must be followed. Synchronicity is the notion that all process transitions for every TE in a given set of TEs happen simultaneously, and thus atomically. All data flow defined on the TEs in a set is synchronous, i.e. data can flow through conjunctively composed rendezvous, from one TE to another, and such data flow is evaluated in light of the synchronicity. As a result, we can consider that such a system models combinational paths. The task of the scheduler is to allow the system to make progress. It does so by generating a set of TEs. We can call a set of TEs, whose domain is defined on any subset of the set of processes, a schedule for the system if

it fulfills the validity constraints below.

1. Process Constraint: There can exist only one scheduled TE per process from the current process state.

2. Participation Constraint : Every scheduled TE must participate in all rendezvous that are annotated on it

3. Role Completion Constraint : Every rendezvous that is seen in the schedule must have exactly 1 TE corresponding to each of its roles

4. Contention Free : There should be no contention of shared variables (i.e. two TEs in the same schedule cannot write to the same register)

5. Condition Satisfaction : Every condition present on every scheduled TE must be true

Certain properties of schedules are of interest. Two smaller schedules are said to be compatible if the union of the two sets also satisfies all validity constraints. This is easy to examine if the process domains of the two smaller schedules do not intersect. In such a case, the schedules affect independent sections of the system. The union of these two schedules will not violate any constraints. The schedule formed by the union is said to be composed. On the other hand, if two schedules satisfy the validity constraints, but the union of their set does not, they are referred to as incompatible. A schedule can be called minimal if there are no schedules that can be composed to create it. Thus, there is no smaller subset of a minimal schedule that can be a schedule chosen instead of the given set. Similarly, we can define properties of any good communication scheduler

1. Scheduler Validity : Any schedule produced by the scheduler should be valid (i.e. satisfy the rules previously defined)

2. Live-ness : If there exists at least one non empty schedule, the scheduler should produce a non empty schedule.

The practical effect is that the scheduler is an external mechanism for removing non determinism in the model. A scheduler then can be defined as an algorithm, implementing a particular policy, which on being given as input a set of processes, their state and produces as output a schedule, and satisfies the general scheduler properties.

4.3 Policies

Unfortunately, as shown in [Joung and Smolka, 1996] the problem of creating such a scheduler that must handle variability in the presence of free composition of disjunction and conjunction, implementing a general global policy is an NP hard problem. In practical problems, such complexity is rarely fully reached. Further, the choice of scheduling policy affects the complexity of the scheduler implementation. As a result, heuristic based algorithms, or pruning based algorithms are capable of producing significant speedups in practical models. Further, we will have a short discussion of possible policies and analyze them using the graphical framework in order to better understand their effect on the complexity of the scheduler.

Consider a process network consisting of rendezvous connecting communicating processes, with each rendezvous having some non negative integer weight associated with it, representing its importance. Each schedule will then have a weight, which is the sum of the weights of the rendezvous activated in that schedule. In this context, we can examine several policies for the choice of a scheduler. One possible policy would be to select the valid schedule which contains highest total weight. This policy, called the Global Weight Optimal (GWO) policy, appears simple, but has significant complexity due to the global nature of the policy. In order to ensure global weight

optimality, different combinations of schedules must be tested. As a simple example, if we consider a system with 3 valid schedules S, S' and S", with weights such that S > S' > S", then simply selecting S as the optimal schedule may not be enough. If S' and S" are compatible, but S is incompatible with the others, then it is possible that a new schedule S"' formed as composition of S' and S" may have a higher weight than S.

Another possible policy would be to guarantee that if two incompatible minimal schedules are valid, the higher weight schedule would be chosen. In such a case, we do not need to worry about considering all compositions of schedules in generating new schedules. Furthermore, since the weights are known before hand, and the run time evaluation of compositions is not being done, the problem of run time selection becomes easier. Thus, the policy uses local weights to create a static order a priori, which is used in the dynamic scheduling phase to determine the final schedule. Such a policy, referred to as Static Local Weight Ordering (SLWO) policy, represents a relaxed superset of the GWO policy, that simplifies the scheduler implementation.

Chapter 5

Graph Based Tools

In this chapter, we describe the novel graph based approaches that we created to create a communication scheduler. We start by presenting the concept of a Transition Relation Graph (TRG), and examine how we generate a set of simultaneous communication patterns, called Minimal Candidate Schedules(MCS). We then describe the Occurrence Relation Graph(ORG), which used to identify all communication patterns in the system. We examine how scheduler policies are mapped into the graphical algorithms and affect the generation of the final communication scheduler.

5.1 Graphical Analysis of Process Networks

As discussed in Chapter 2, other rendezvous based approaches avoid using a scheduler, instead choosing to restrict the expressivity of the language to remove nondeterminism from the system. In order to understand and visualize the complexity of the problem the scheduler must solve, it helps to examine the possible ways the system may progress. This can be done by finding and examining relationships between all possible valid minimal schedules. In order to perform this, we propose the following two step algorithm. First we convert the input process network into a Transition Relation Graph (TRG), which relates TEs and rendezvous and captures various compositions in the original system. Based on the TRG, we compute a new graph, the Occurrence Relation Graph (ORG), which relates all possible minimal candidate schedules. This ORG can then be used to analyze the original system and the effects of various

policies.

5.2 Transition Relation Graph (TRG)

The purpose of the transition relation graph is to establish occurrence relationships between different state transition edges. There exist two types of TRG vertices, transition vertices, which represent TEs from the original system; and rendezvous vertices, which represent rendezvous. A transition vertex is annotated with a set of rendezvous labels belonging to the corresponding transition edge. Each rendezvous vertex is annotated with a rendezvous label. In order to capture the relationships between TEs, we define two kinds of TRG edges, a related edge (represented graphically as a solid line) and mutual exclusion (ME) edge (represented by a dashed line). ME edges can be divided into two basic types. Deterministic ME (DME) edges are those that can be resolved using run time information of the system. For example, when two TEs leaving the same state have mutually exclusive data conditions, we create a DME edge. Nondeterministic ME (NME) edges on the other hand require run-time decision by the scheduler for one of the transition edges to be selected. Such edges are typically introduced due to variability and disjunctive composition. An NME edge can also be introduced when both transition edges write to the same variable. Such edges help to prevent race conditions in the system. In the TRG, we prefer DME edges. Thus, if a pair of transition vertices is related by an NME edge, but can be resolved using run time information, the NME edge is converted to a DME one.

Graphically, the TRG is the union of a bipartite graph containing solid edges connecting rendezvous vertices and transition vertices, and a set of ME edges between transition vertices.

To construct a TRG for a system, we first create transition vertices for all transition edges in the system, and rendezvous vertices for all rendezvous. As a small

optimization step to reduce the number of vertices, we merge sets of transition vertices that share the same set of rendezvous labels but are at different source states in the same process. Since a process can only be in one state at a given time, the actual transition edge that this merged vertex represents can be fully resolved at run time. We then create the edges in the TRG.

1. For every rendezvous labeled on a transition vertex, draw a solid edge between the corresponding rendezvous vertex and the transition vertex.

2. If two transition vertices share a common role of a rendezvous, draw an NME edge between them. This captures the competition due to variability.

3. If two transition vertices share a common source state, draw an NME edge between them. This edge captures the choice due to disjunction.

4. If two transition vertices from two processes write to the same variable, draw an NME edge between them. This prevents race conditions.

5. If two transition vertices from two processes have mutually exclusive data conditions, draw a DME edge if none exists, or convert the existing NME edge to a DME one. This allows us to distinguish between ME relationships that can be resolved at run time and those that cannot.

As a final pass, after all the edges are created, if any NME edge connects transition vertices that belong to the same machine, but have different source states, the NME edge is converted to a DME one.

We use the process network in Figure 2·1 as an example. For the clarity of description, we make the reset edges explicit, and mark each transition edge with a unique integer, as shown in Figure 5·1. The TRG diagram will refer to these numbers.

We start by converting all the state transition edges into TRG transition vertices and all the rendezvous into TRG rendezvous vertices. This resulted in 16 transition

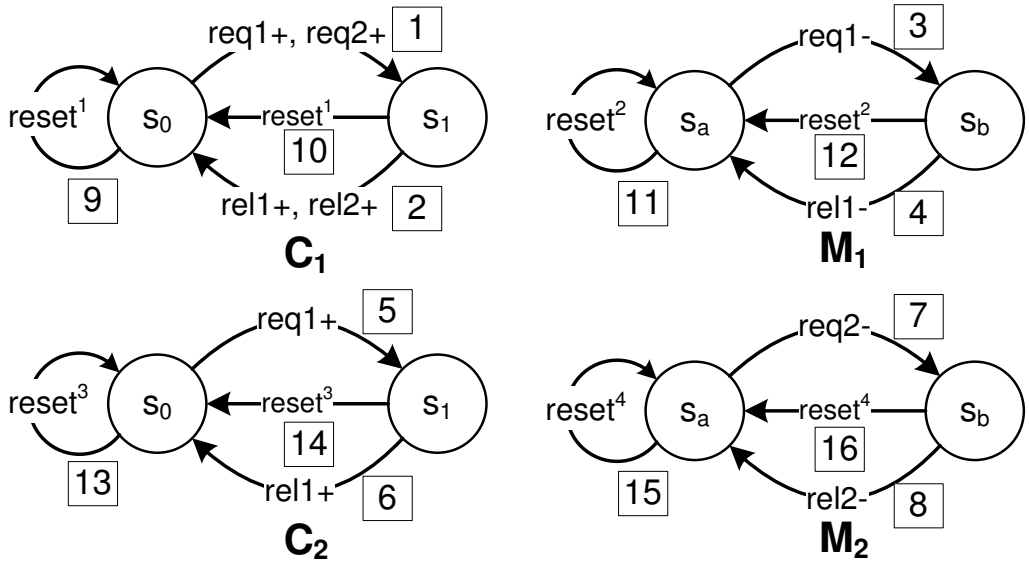

Figure 5·1: Annotated Example Process Network

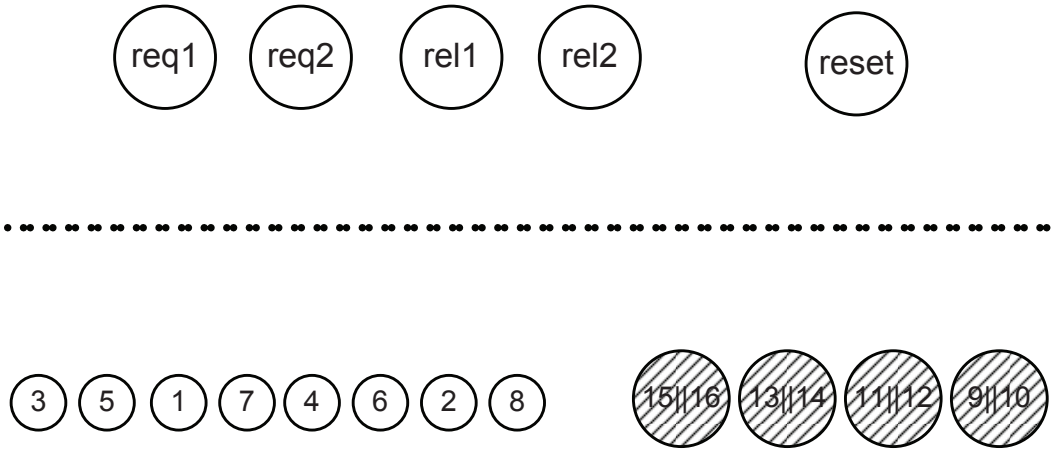

Figure 5·2: Vertices in the TRG

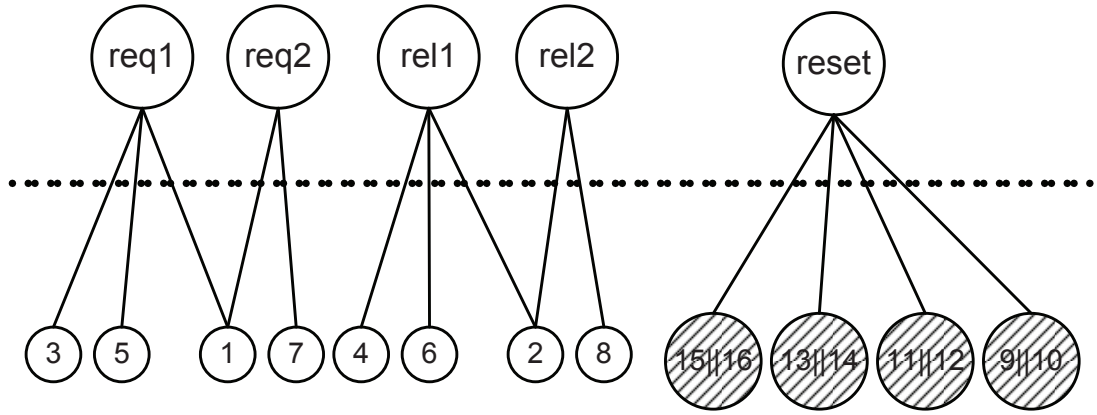

Figure 5·3: Adding Solid edges in the TRG

vertices and 5 rendezvous vertices. As our first optimization step, we merge the two transition vertices labeled with reset of every process. This results in all the vertices shown in Figure 5·2. The merged transition vertices are shaded diagonally. In effect, we have a graph that can be divided into 3 types of vertices : Transition vertices, Merged transition vertices and rendezvous vertices. The merging process helps us reduce the size of the graph by reducing the number of nodes in the system. This helps simplify the succeeding steps.

The next step is the addition of solid edges in the TRG, shown in Figure 5·3. In this step, we create solid edges between the transition vertices and the rendezvous vertices who share a common rendezvous. For example, since we can see that Transition Edge 3 contains a "req1-" role, we can create a solid edge between the vertex labeled 3 and the req1 rendezvous vertex. Similarly, since TE 1 contains the conjunction of "req1+" and "req2+" roles, we create two solid edges from the vertex labeled 1 to the req1 and req2 rendezvous vertices. Thus, we can note that for a transition vertex, the number of solid edges from the vertex is the degree of conjunction of rendezvous present on that TE in the original system. For a rendezvous vertex, the solid edge degree simply represents the total number of roles of that rendezvous in the system.

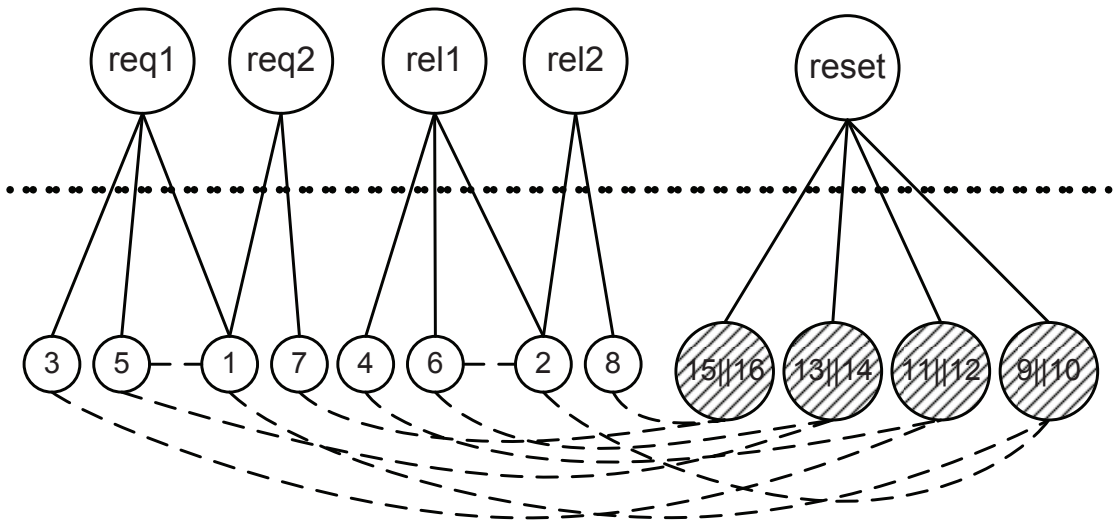

Figure 5·4: Final TRG

Note that since a rendezvous has at least 2 roles, and solid edges do not preserve the role information, the solid degree of a node cannot be directly correlated to the amount of nondeterminism present in the rendezvous.

Then, we create ME edges for this TRG, by examining pairwise all transition vertices. Any two transition vertices can be related using mutual exclusion either deterministically (DME) or nondeterministically (NME). The main cause of the nondeterminism arises from the input system description from the designer. As a result, the nondeterminism in this graph is a direct transformation of the original system. There are 3 main sources of nondeterminism. Firstly, nondeterminism caused by the choice of exactly how to fire a rendezvous. This nondeterminism is easily identified as two transition vertices sharing an identical rendezvous role label. For our example, we can consider the vertices 1 and 5, both of which posses a common "req1+" role label. As a result, in the TRG, we must introduce an NME edge between the two transition vertices. A second source is the presence of disjunction, or choice within a process. This is captured by examining the source states for the TE corresponding

to the given transition vertices and creating NME edges between them if the source states are identical. There is a caveat here, in that if there additionally exists a guard condition that can be statically shown to be deterministically mutually exclusive, this edge can be created as a DME edge instead. In this phase of the TRG construction, determinism trumps non determinism. To clarify, if there exist a pair of transition vertices that would be related as nondeterministically mutually exclusive (NME), but due to statically exclusive guard condition will always provably be resolved, we can a priori resolve the nondeterminism and promote the NME edge to a DME one. This promotion is responsible for the reduction of scheduler nondeterminism and has consequences for the ORG construction phase, as will be discussed later. In this example, if we examine the pair of transition vertices 1 and 9|10, we can see that they share a common source state of S_0 in process C1. As there is no static guard condition, we can draw a NME edge between the two vertices. The final source of NME edges is from race conditions on shared variables. This is captured by examining the pair of transition vertices to see if their semantic actions write to the same shared variables. In this case too, if there is a precluding source of determinism, we can convert the NME edge into a DME one. In this example, there are no such edges, but it would not be difficult to see why this is necessary. Adding all the ME edges gives us the TRG shown in Figure 5·4.

The TRG finds relationships between all Transition Edges in the system. Doing this allows us to begin examining possible valid minimal schedules in the system. We can see that each solid connected TRG component represents one possible schedule for the system. The algorithms for doing this are discussed in the remainder of this section.

From the construction algorithm, it is easy to see that the size of the TRG is well bounded for a given system description. The number of vertices in the TRG is

always no higher than the sum of the number of transition edges and the number of rendezvous in the system. The number of solid edges in the TRG is bounded by the total number of rendezvous labels on all transition edges. However, the number of NME edges added due to variability and disjunction are quadratic with respect to the degrees of variability and disjunction. In practical systems, these sources of nondeterminism are usually limited. The number of DME edges is dependent on the model and is hard to quantify without knowledge of the actual system. However, since DME edges help us eliminate occurrence choices, they simplify the analysis of the system. Thus, a large number of ME edges at this stage is beneficial for our approach. Additionally, we also note that all TEs belonging to the same process form a ME clique. In this clique, TEs that are nondeterministic (via disjunction from some state) are related via NME edges, and all other edges are DME.

5.3 Minimal Candidate Schedules (MCS)

A minimal candidate schedule(MCS) is a set of transition vertices that form a valid schedule for the system and are minimal, i.e. there is no nonempty subset of the MCS which is also a valid candidate schedule. While further discussion of the MCS is present in Chapter 4, it is easy to examine the effect of the MCS in graphical terms. As a refresher, an MCS must satisfy the following properties :

1. Process Constraint: There can exist only one scheduled TE per process from the current process state.

2. Participation Constraint : Every scheduled TE must participate in all rendezvous that are annotated on it

3. Role Completion Constraint : Every rendezvous that is seen in the schedule must have exactly 1 TE corresponding to each of its roles

4. Contention Free : There should be no contention of shared variables (e.g. two TEs in the same schedule cannot write to the same register)

An important fact to keep in mind here, is that by definition, an MCS does not have any requirement on the condition satisfaction criterion, the idea behind the MCS is that the schedule can become valid. Thus, the condition may be dynamically satisfied, i.e. satisfied at some future run time iteration.

In order to find all candidate schedules in the system, we need to examine the TRG that has been created. Since the TRG is statically generated, TRG vertices with data conditions will create candidate schedules due to the lack of run time information to check the data conditions. This candidate schedule will have to be evaluated at run time to check if all the validity constraints are met for it to become a valid schedule. If so, the candidate schedule is said to be enabled. An MCS is observed to be a solid-edge connected sub-graph of the TRG which satisfies the following three conditions.

1. For every transition vertex in the sub-graph, all its solid edges must be included.

2. For every rendezvous vertex in the sub-graph, we must include as many solid connected neighbors as it has roles.

3. For every sub-graph that is selected, there should be no two vertices connected in the original TRG by an ME edge.

Together, these three rules help find all minimal candidate schedules in the system. The first rule ensures that all rendezvous on a transition edge are included, i.e. that the set of transition vertices meets the Participation Constraint. We can further observe that this rule encapsulates complexity from conjunctive composition, as that would cause transition vertices to have more than one solid neighbor.

The second rule ensures that the sub-graph contains as many transition vertices as the roles the rendezvous has, i.e. the Role Completion Constraint for the set of transition vertices is met. Now, we must note that this step may not be trivial, as it is possible that the rendezvous can satisfy a role using multiple transition vertices.

The third rule ensures that all transition vertices in the sub-graph are compatible. During the creation of the TRG, we note that all TEs belonging to one process are related either via DME or NME edges, as a result, we satisfy the Process constraint. We can also note that the creation of NME edges during the TRG construction ensures that the third rule will also ensure that candidate schedules are free of resource contention, variability and disjunction. As we can see, the only remaining property to be checked for validity of the candidate schedule is the condition satisfaction. However, this might depend on run time information, and cannot be done a priori.

The algorithm for the constraint based computation of the MCS is based partly on classical backtrack based approaches, with some heed paid to dynamic programming and early pruning. This algorithm, referred to as the wavefront algorithm depends on the definition of a few concepts.

A *wavefront* is described as a set of TRG vertices of the same type that represent the active vertices that are being processed at each phase of the algorithm. Each wavefront is used to generate its successor, starting from an initial seed point. A terminal wavefront is a wavefront that has no successor. This history of wavefronts is called the *path* of a wavefront. Effectively, the path of a terminal wavefront will be a solid connected component.

The algorithm can be described in simple terms as follows. The problem is broken into solid connected components, temporarily ignoring the ME edges between them. Then, we perform a depth first search of the TRG beginning at an arbitrarily selected initial transition vertex, but in sets of vertices called wavefronts. At each step in the

computation of the children, we avoid the sets that will cause the component to be an invalid schedule, which is easily seen as a ME edge in the TRG. Once we cannot make any further progress, we must stop, and if some basic conditions are satisfied, add the entire set of transition vertices in the path into a MCS for the system. Finally, in order to ensure all possible MCS are captured, we must repeat the algorithm with the initial point set to each of the ME neighbors of the original choice.

1. Divide the graph into solid connected components
2. Begin at any Transition vertex, as the initial wavefront
3. Mark all vertices as seen
4. For a wavefront containing transition vertices, compute the next wavefront as all solid connected unvisited Rendezvous vertices
5. For a wavefront containing rendezvous vertices, compute a set of next wavefronts such that there are no mutual exclusion edges between them and vertices already in the path
6. Repeat until the wavefront becomes terminal
7. If the path of the terminal wavefront satisfies the conditions, add it to the list of MCS
8. Unmark all the vertices in the current wavefront and return to the previous step in the path
9. Repeat for all mutual exclusive neighbors of the original starting node
10. Repeat for all disjoint components

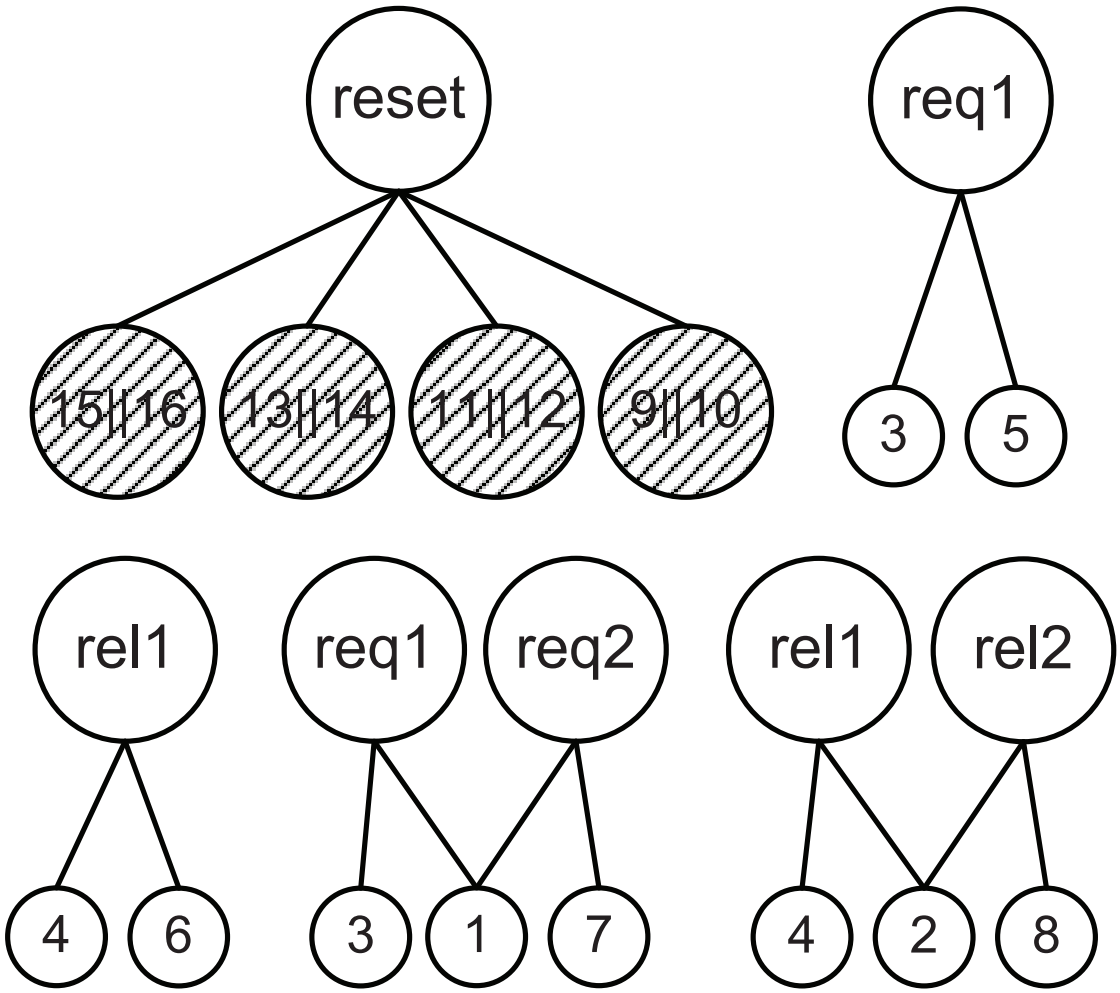

Figure 5·5: Minimal Candidate Schedules in the system

For the example network in Figure 2·1, we can examine the wavefront algorithm in action. Let us say we begin by examining the TRG. We can see that there are 3 solid connected components as seen in Figure 5·3. Let us begin with the component containing the transition vertices 3,5, 1 and 7. Let us begin with the choice of vertex 3 as the initial vertex for the first wavefront. Thus, we compute the first wavefront to contain the vertex 3, and mark 3 as visited. The next vertex must contain req1 and as it has not been visited and is compatible with the existing path, we create the next wavefront containing req1. We now mark req1 as seen, and since this vertex is a rendezvous, we examine the number of roles required. As req1 is a biparty rendezvous, we need to determine how to compute the next wavefront. We can see that there exist 2 solid neighbors of req1, 5 and 1 which are compatible with the path so far. However, 5 and 1 are NME related. As a result, there are 2 possible wavefronts we can progress along. The first wavefront will contain only 5. This wavefront is terminal as there are no unvisited solid neighbors. Additionally, the conditions for role satisfaction are met by the path $\{3, req1, 5\}$. Thus, we can create the MCS $\{3, 5\}$. Now, we unmark node 5 and return to the previous wavefront in the heirarchy. At this point, we make the second choice, i.e. the next wavefront as 1. We mark 1 as seen, and note that there is only one way to make progress, i.e. wavefront to req2 and then to 7. As a result, the path for the terminal wavefront at transition vertex 7, $\{3, req1, 1, req2, 7\}$ is created as a new MCS $\{3, 1, 7\}$. Now, all these wavefronts have been simple, i.e. have contained only one vertex. If the choice of initial vertex had been transition vertex 1, then the algorithm would have proceeded differently. The first wavefront would have been 1. The next wavefront would have been $\{req1, req2\}$. The successor wavefront to this would be the wavefront 3,7. As this is a terminal wavefront, and the path meets the conditions, $\{1, req1, req2, 3, 7\}$ would be created as a new MCS. Note, that this is the same as the second MCS created from the choice of initial vertex as vertex 1. Now,

since there was a ME neighbor of the source vertex, i.e. 5, the algorithm is repeated starting at that vertex. For this case, the next wavefront is $\{req1\}$, which in turn has the terminal successor wavefront $\{1\}$, giving rise to the MCS $\{5, req1, 3\}$. Thus, the choice of the starting point does not affect the final result, but does affect the order of the generation of MCS. The final MCS are shown in Figure 5·5.

An important fact to note is that the wavefront algorithm divides the TRG into components based solely on solid edges. Thus ME edges between solid components are ignored for this phase of the scheduler construction. However, these edges must not be deleted as they represent important information about the nondeterminism the scheduler must resolve. The information about these edges thus is still preserved in the TRG and will be used in the next step. The reason we can ignore them in this step is that the ME edges between solid connected components are effectively edges that affect the run time decision of arbitration between schedules, whereas ME edges that exist within a solid connected component are the edges that determine whether two edges can exist in a schedule and are thus static. Thus, we can say that ME edges that affect the static portion, i.e. whether a set of TEs creates a schedule is always local to a solid connected component. ME edges that affect scheduling nondeterminism on the other hand (i.e. choices between schedules) can be either local (i.e. contained within a solid connected component) or remote (i.e. between solid connected components).

5.4 Occurrence Relationship Graph (ORG)

The Occurrence Relationship Graph relates all minimal schedules. Each vertex of the ORG is a unique minimal candidate schedule for the system. Each ORG vertex can also be assumed to carry a weight, which is the sum of the weights of its component state transition edges. The vertices of the ORG are related via ME edges of different

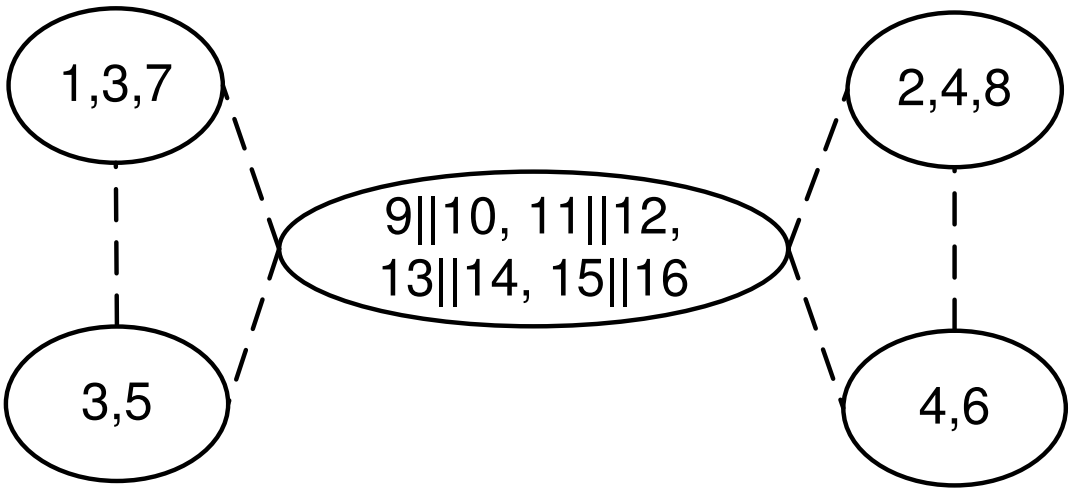

Figure 5·6: Occurrence Relationship Graph

types. Unrelated vertices can be considered to be compatible schedules and thus capable of occurring together.

We can construct the ORG by first determining all candidate schedules in the system via enumeration. We then relate the candidate schedules by carrying the NME edges in the TRG to the ORG. Thus, if two schedules contain TRG vertices that are related in the TRG via a NME edge, we can draw a ME edge (denoted as a black dashed line) between the two ORG vertices. DME edges from the TRG do not need to be carried into the ORG as they can be resolved at each run time iteration by the state of the machine or the dataflow conditions for a given schedule. Thus, they represent relationships which do not need to be arbitrated by the scheduler.

If two schedules can be multiply related, i.e. they contain one pair of TRG vertices related by DME and another by NME, then the NME relationship dominates the DME. Thus, the resulting ORG vertices will remain connected via an NME edge and will require the help of the scheduler at each run time iteration.

In the case of the system in Figure 5·1, we can walk through the generation of the ORG. Let us consider pairwise all MCS created using the wavefront algorithm.

In this situation, we can draw NME edges between any 2 MCS that were so related in the original graph. If we consider $\{1, 3, 7\}$ and $\{3, 5\}$, we can note that the TRG shows a NME edge between 3 and 5. As a result, the two MCS are ME related in the ORG. Similarly, $\{2, 4, 8\}$ and $\{4, 6\}$ are ME related in the ORG. Finally, the MCS corresponding to the reset rendezvous, i.e. $\{9|10, 11|12, 13|14, 15|16\}$ is ME related to all other MCS from the relationship seen in the TRG. In the case of unrelated MCS, such as $\{3, 5\}$ and $\{4, 6\}$ there is no relationship between the MCS as there is no mutual exclusion between the TRG vertices. The final ORG is shown in Figure 5·6. For the sake of simplicity, each ORG vertex is annotated only with the numbers of the TRG vertices in the related candidate schedule

An important fact is that the ORG only contains one type of edge : Nondeterministic Mutual exclusion. Thus, every edge in the ORG represents nondeterminism in the system. Additionally, unrelated MCS in the ORG are assumed to be concurrent, as they may be active independently. The ORG is the graphical representation of the nondeterminism present in the system. In the case of the formal representation of the scheduler is explained in Chapter 6. As a way of linking the two explanations, the MCS nodes together represent the input set. The Acceptable set is the set of all subgraphs of the MCS such that no two MCS are mutually exclusively related.

5.5 Policy Implementation

This work proposes to use a scheduling algorithm based on the ORG. Any such algorithm must satisfy the live-ness and validity criteria proposed in Section 4.2. One way such an algorithm would work is to implement the policies in such a way that the general scheduler properties are met by the policies themselves. In this case, the scheduling algorithm consists merely of providing inputs to and using the outputs of the scheduling policy to update the global state. This is the scenario being

considered in this work. The role of a scheduler is to resolve nondeterminism in the system, and select a set of transitions to activate. Since transitions are grouped into minimal candidate schedules in ORG, and run-time nondeterminism are preserved in the ORG as ME edges, the ORG is a suitable tool for the scheduler to rely on. To simplify the task, the scheduler can first prune the ORG by excluding candidate schedules that are invalid at the current global state. As a process can only be in one state at a time, all candidate schedules involving transition edges that are not related to the current states can be eliminated from consideration. Similarly, since each minimal candidate schedule is free of nondeterminism within itself, the complete data flow on the transition edges are deterministic. Thus, all Boolean conditions can be evaluated. If one transition edge has a false condition, the ORG vertex is disabled. The validity of candidate schedules is determined using this run time information. For example, in Figure 2·1, when the variable timeout is false, all ORG nodes containing that transition edge are disabled.

5.5.1 Global Weight Optimal Policy

The Global Weight Optimal (GWO) policy is the policy where the scheduler selects a schedule, which may be composed of several minimal schedules, such that the weight of the selected schedule is the highest possible. We need to compute the maximum weight set of enabled schedules at each run time iteration. We perform this by first encoding the ORG into a tree based structure statically, and then using run time information to selectively enable or disable certain candidate schedules. A single pass evaluation is then used to compute the next valid schedule. In order to compute the best set of minimal schedules to be chosen, we can create a decision tree. This tree will contain 3 types of nodes.

1. Internal decision nodes represent points where a decision will be made whether a particular schedule is selected or not.

2. Internal join nodes connect schedules that are independent.

3. The leaf nodes of the tree are schedules.

BUILD(G, V_{pin})
if $num_vertices\,(G) = 1$ **then**
  l =new leaf node (V(G))
  return l
else if G is disconnected so that $G = \bigcup G^i$ **then**
  j =new join node
  for all G^i **do**
    add BUILD(Gi, V_{pin}) as child to j
  end for
  return j
else
  find $v \in V\,(G)$ of maximum ME degree st $v \notin V_{pin}$
  d =new decision node (v)
  $G' = G \setminus$ ME neighbors of v
  $G'' = G \setminus \{v\}$
  add BUILD(G', $V_{pin} \cup \{v\}$) as taken child to d
  add BUILD(G", V_{pin}) as untaken child to d
  return d
end if

The algorithm starts with an input ORG. It adds all disconnected components as children of a join node. For each connected component, it selects the vertex with the highest ME degree. It then creates a decision node in the tree around this vertex. The two children of the decision node represent the two possibilities, one child, called the taken branch, where the vertex is present, and as a consequence, all ME neighbors

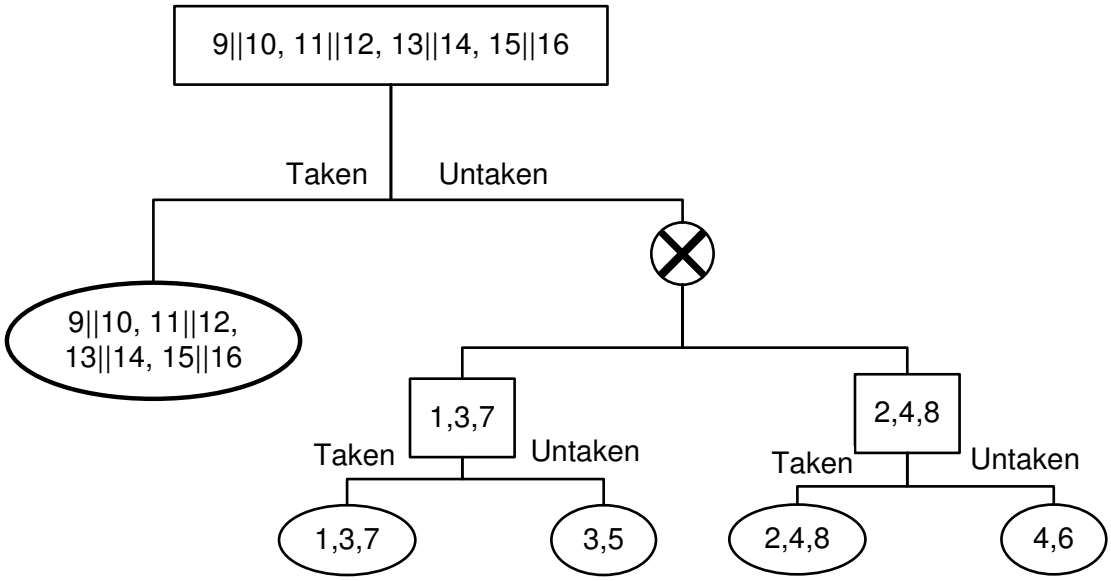

Figure 5·7: Tree for GWO policy scheduling

are absent. And the other, called the untaken branch, where the vertex is absent. In both cases, new subgraphs of the ORG are generated and the partition process is continued until the ORG has exactly one vertex remaining. When this happens, the vertex is added to the tree as a leaf node.

The algorithm is described in pseudo code above. It accepts G=(V,E) and Vpin, where G is an ORG subgraph, V is the set of its vertices and E is its set of edges, and Vpin is a set of pinned vertices. Initially, G is the entire ORG and Vpin is empty. As the algorithm runs, G empties while Vpin increases in size. Since the algorithm implementing this policy is reductive, i.e. every operation reduces the size of G, the algorithm will terminate after execution. In the case of the example in Figure 5·1 the decision tree as shown in Figure 5·7. The internal nodes represented by a square are decision nodes. The internal nodes represented as a circle with a cross though them are join nodes. The ovals are leaf nodes.

The per iteration selection task then consists of first enabling all candidate schedules based on the current global states and condition satisfaction rules. Then, we can compute the weights in a bottom up manner. Decision nodes will add the weights of their children. At pick nodes, we implement our policy by picking the child which has the higher weight. Thus, at the root of the tree we obtain the final set of schedules which satisfy the GWO policy for a given iteration. The software implementation is an algorithm that accepts the statically generated tree (T) and the global state (S) as inputs and produces a schedule as its output. It runs the function enable that enables the TRG vertices, and, by implication, the ORG vertices, based on the current state and the value of the guard conditions. The getWeight helper function returns the sum of weights of all enabled schedules in a subtree rooted at a given vertex.

```
GWOScheduler (T, S)
enable(T,S)
return traverse(root(T))
traverse(v)
if v is a normal leaf then
  return {v}
else if v is pruned then
  return
else if v is join then
  for each child Ci of v do
    result = traverse(Ci)
  end for
  return result
else if v is decision then
  result_t = traverse(C_taken)
  result_u = traverse(C_untaken)
  taken_wt = getWeight(result_t)
  untaken_wt = getWeight(result_u)
  if taken_wt > untaken_wt then
    return result_t
  else
    return result_u
  end if
end if
```

Compared to the SLWO policy discussed next, the GWO policy selects at a global scope and this has more complexity. This form of scheduling is more suitable to

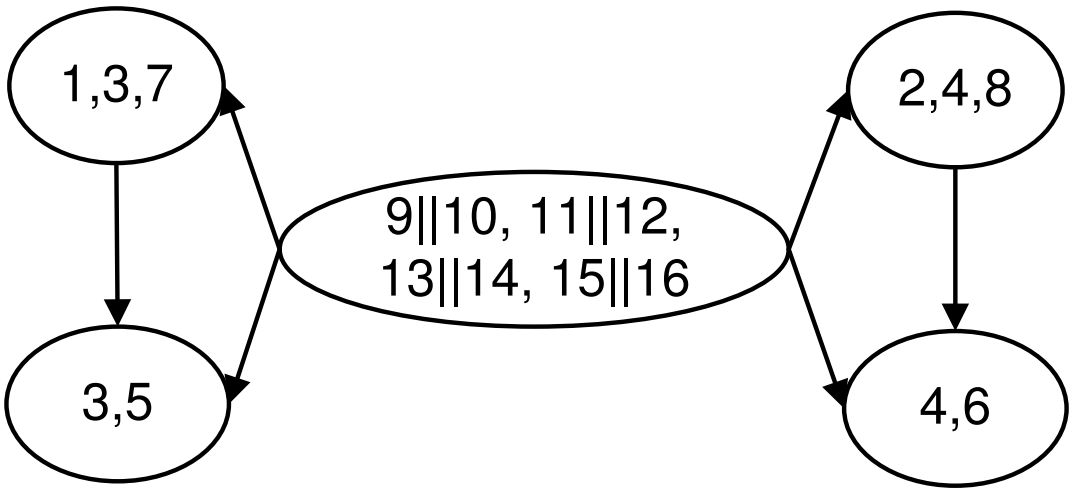

Figure 5·8: Directed Acyclic Graph for SLWO policy scheduling

smaller process networks.

5.5.2 Static Local Weight Ordering Policy

Static Local Weight Ordering is a much easier policy to implement. In this case, the scheduler can fully prioritize all vertices in the ORG according to user provided hints (e.g. weights for rendezvous, or for transition edges). The scheduler will always pick a maximum set of ORG vertices, subject to the following rule: if a ME edge exists between two enabled vertices, pick the one with higher priority. In the example ORG in Figure 5·6, when all processes are at their left states, the right two vertices are disabled due to the current states. If the vertices are prioritized based on the number of transition edges included in each, the middle vertex has the highest priority, followed by vertex (1,3,7). If the variable timeout is true, then the middle vertex will be selected. Due to the ME edges, the left two vertices are disabled. If timeout is false, the (1,3,7) vertex is selected. We can convert each ME edges in ORG into directed edges from the higher priority vertex to the lower priority one. The whole ORG then becomes a directed acyclic graph (DAG) as shown in Figure 5·8.

The per iteration selection task then consists of first enabling all candidate schedules based on the current global states and condition satisfaction rules. Then, we suppress all schedules according to the directed edges in the ORG. Thus, an ORG vertex with a higher priority will supersede one with a lower priority. In the example above, if all schedules are enabled, then the (9||10, 11||12, 13||14, 15||16) ORG vertex, corresponding to the reset rendezvous will disable the others. If the (1,3,7), (3,5), (2,4,8), (4,6) schedules were the only ones enabled, then the priority rules would cause (3,5) and (4,6) to be disabled. Thus (1,3,7) and (2,4,8) would be selected for occurrence. As each directed edge determines how to disable NME related schedules, the resulting scheduler will always remove all non determinism from the system.

Chapter 6

Formal Notation

In this chapter, we will examine the current ways finite state machine based languages are formally represented and extend the definitions to include the use of rendezvous as a communication mechanism. Further, we will develop use the notation on the state machines in Lyra and examine their effects on the formal notation of the graphical tools developed in Chapter 5. Finally, we will examine the formal notation for a complete Lyra model, including the scheduler.

6.1 Basic Concepts

Finite State Machines (FSMs) are a well studied branch of digital design. From their inception, they have been defined using formal mathematical models. Informally, a Finite State Machine is a process containing a set of states and transition between states. At any given instant in time, the FSM is in exactly one state. To make progress, the FSM computes the next state it should transition to, based on its definition and atomically activates that transition, updating its state and producing some output. Classically, a Finite State Machine is defined as a 6 tuple [Hachtel and Somenzi, 1996]

$$\langle I, S, \delta, S_0, O, \lambda \rangle$$

where:

I is the input alphabet, i.e. set of input values

S is the set of states

$\delta : S \times I \rightarrow S$ is the next state function

$S_0 \subseteq S$ is the set of initial states

O is the output alphabet

$\lambda : S \times I \rightarrow O$ is the output function

Here, we are making a few assumptions. Firstly, this representation is that of a Mealy type machine, where the output is a function of the state and inputs. While there are equivalent transformations from Mealy to Moore machines, it is easier to deal with Mealy machines as the choice of state transition affects the output in cases of nondeterminism. The next assumption is that both I *and* S are finite, non empty sets. This follows from the basis of definition of state machines. In this section, as previously, we will be graphically presenting state machines as state transition diagrams. Furthermore, we are assuming that the FSM being described is deterministic and completely specified.

While this a sufficient description of a standalone FSM, for practical applications, we need to extend the definition of an FSM to allow for transitions and local variables. Here, we can define an Extended Finite State Machine (EFSM) as follows [Lee and Yannakakis, 1996] :

$$\langle I, S, T, V, S_0, O \rangle$$

where:

T is the finite set of all transitions

V is the finite set of all variables

Each transition t in T is defined in turn by the tuple :

$$\langle S_{src}, S_{dst}, I_{in}, O_{out}, C_t, A_t \rangle$$

where:

$S_{src} \in S$ is the source state for the transition

$S_{dst} \in S$ is the destination state for the transition

$I_{in} \in I$ is the input value

$O_{out} \in O$ is the output value

C_t is condition predicate on variables for the transition. $C_t(V)$ is either TRUE or FALSE, depending on whether the condition for the transition are satisfied.

A_t is the action on variables for the transition. $A_t(V)$ will assign a value to each element in the set V

As we note, EFSMs and FSMs differ in some crucial ways. Firstly, EFSMs now have the notion of local variables, and correspondingly semantic actions and guard conditions. The local variables act as data storage within a processes, allowing the modeling of real hardware systems, where a process needs to retain state information and data. Semantic actions, defined as $A_t : V \rightarrow V$ are basically present to allow the assignment of new values into the local variables. In general, we can assume that $V_i = A_t(V_i)$ where $A_t(V_i)$ is an arbitrary function of V_i. More interesting is the case of the condition predicate. We can informally define the predicate of a transition as a "guard condition" or, the condition, which allows the transition to be used as a valid transition in the EFSM if and only if $P_t(V) = TRUE$. As a further definition, we can define $X_{P_t} = \{V_i : P_t(V_i) = TRUE\}$ as the Valid Variables of transition t.

We changed the way transitions are defined. We could equivalently describe FSMs as the 5 tuple:

$$\langle I, S, V, S_0, O \rangle$$

where:

$$T = \delta \cdot \lambda$$

So far, we have been considering deterministic FSMs, i.e. FSMs where there is exactly one state for any given pair of current state and input. In effect, this implies that there are no two transitions that share the same source and input value but have

different destination states. We can express this as

$$if \left(t_{i_{S_{src}}} = t_{j_{S_{src}}} AND t_{i_{I_{in}}} = t_{j_{I_{in}}} \right) \Leftrightarrow i = j$$

or alternately, we can say that for a given tuple (s, i) such that $s \in S$ and $i \in I$ the δ function, presents a single element s' where $s' \in S$.

For nondeterministic FSMs, we do not have this restriction. Thus, it is possible for multiple transitions to have the same source state and input value, but have different destination states. The addition of local variables does not affect this definition except to add an extra degree of flexibility. Thus, for nondeterministic EFSMs, multiple transitions may share the same source state and input alphabet, but may vary in one or more of : destination state, condition predicate, or semantic action. Another difference between a nondeterministic EFSM and a deterministic EFSM is in the property of the condition predicate. In deterministic EFSM, the condition predicates of any two transitions must be mutually exclusive. This follows from the definition of a deterministic EFSM, i.e. for any given state and input, there must exist only one transition that can be used. For a nondeterministic EFSM, the condition predicate may not be mutually exclusive.

6.2 Lyra Formalization

As might be intuitively obvious, Lyra models correspond closely to the nondeterministic FSMs previously proposed. The main difference, of course, is in the use of Rendezvous, the synchronous communication construct[Edwards et al., 2001]. In general, a Lyra system can be defined as the tuple :

$$\{P, R, GI, GO\}$$

where:

P is the set of processes

R is the set of rendezvous

GI is the set of global data inputs

GO is the set of global data outputs

As a quick note, the inputs defined as GI and GO are purely signal based. Furthermore, the inputs in GI cannot directly write to a register. They must be read by a semantic action on a transition that then performs the write to a state holding element. The outputs in GO are either the output of a register, potentially involving additional combinational logic, or directly generated as combinational functions from GI. The primary item to note is that no element in GI or GO is a rendezvous.

Each process p in P is similar to an EFSM and can be described as :

$$p = \langle I, S, T, V, S_0, O, DP \rangle$$

where each term retains its meaning from the EFSM description. In this case, the inputs and outputs I, and O, correspond to the signal inputs and outputs for the process. In addition DP refers to the combinational datapath present in the process. The datapath produces an output at every iteration and is only allowed to read from local variables. The output values are thus always a combinational result based on the inputs and the value of the variables. This statement is basically defined as follows : $DP : I \times V \rightarrow O$

Each transition t in T is defined as follows :

$$t = \langle S_{src}, S_{dst}, I_{in}, R_{t_i}, C_t, A_t, R_{t_o} \rangle$$

where:

S_{src} is the source state

S_{dst} is the destination state

I_{in} is the set of input values from the input signals to the process

C_t is the condition predicate for t

A_t is the set of semantic actions notated on t

R_{t_i} is the set of rendezvous role labels that act as inputs for the semantic actions

R_{t_o} is the set of rendezvous role labels that are outputs for the semantic actions

As we can see, the process definition now has multiple possible sources and destinations of data. While the existing sources I and O are used to signify the traditional signal based communication, the rendezvous based communication is primarily shown on the transitions. I_{in} is a set of process input values that are used by this transition. For the condition predicate to be true, we must now add the conditions that all rendezvous in the set $R_{t_i} \cup R_{t_o}$ must be able to commit to occur. Furthermore, the semantic actions are now made up of two portions - the updates to the variables and the updates that must be written to the the elements of R_{t_o}. More formally,

$$C_t = \begin{cases} TRUE: & C_t(v) = TRUE \; \forall v \in V \; \& \\ & \\ & commit(r) = TRUE \; \forall r \in \{R_{t_i} \cup R_{t_o}\} \end{cases}$$

$$A_t = \begin{cases} v = A_t(v) \; \forall v \in V \\ \\ r = A_t(r) \; \forall r \in R_{t_o} \end{cases}$$

A rendezvous is a communication construct that can have n roles. Each role is associated with a unique label. By design, Lyra systems have rendezvous role labels present on transition edges. For the rendezvous to be enabled, a set of valid transitions must be chosen such that exactly one label for each role is present. Also, for the guard condition on a transition to be satisfied, all rendezvous for all role labels present on the transition must be committed to occur. This means that in the remainder of the system, transitions that satisfy this role completion of the chosen set of rendezvous must be selectable. Now, we can formally define a rendezvous r in R as follows:

$$r = \langle RL_0 \ldots RL_n, \delta_r \rangle$$

where: RL_i is a set of Rendezvous Role Labels corresponding to role i. Each element of RL_i is a transition.

δ_r is a function that maps valid sets of transitions that satisfy a rendezvous' conditions.

6.3 Graphical Tool Algorithms

As we have discussed in Chapter 4, the scheduler is a very important part of what makes Lyra unique among rendezvous based approaches. In this section, we will quickly examine the graphical tools used to create the scheduler and examine some bounds on the complexity of the description.

The scheduling algorithm is best expressed as the following steps :

1. Creation of TRG
2. Creation of MCS
3. Creation of the ORG

The basic steps to create the TRG can be summarized as follows :

1. Create a Rendezvous vertex for each rendezvous
2. Create a Transition Edge vertex for each state transition
3. Merge all Transition vertices having the same edge labels, in the same process with different source states
4. Create Solid edges between Rendezvous vertices and the Transition vertices having those rendezvous roles

5. Create Mutual Exclusion edges between two transition edges leaving the same state, two transition edges having a common role label and two transition edges that write to the same vertex

At the end of this, we create a graph $G_{TRG}\,(V_{TRG}, E_{TRG})$ where V_{TRG} is the set of vertices of the TRG. By construction, we know that

$$|V_{TRG}| \leq |R| + \textstyle\sum_{\forall p \in P} |T_p|$$

Furthermore, the number of edges in the TRG can be expressed as a sum of the number of solid edges and the number of mutual exclusive edges. The number of solid edges is exactly equal to the number of rendezvous role labels in the system. The number of mutually exclusive edges is a conditional sum, bounded in the worst case as a fully connected subgraph. A more precise count however is preferable. We can compute the number of transitions leaving the same state and the number of transition edges having the same role label. For now, let us assume there are no transition vertices that write to the same resource. Then we have,

$$|E_{TRG}| \leq \textstyle\sum_{\forall r \in R} \left(\sum_n |RL_i|\right) +$$
$$\textstyle\sum_{\forall r \in R} \left(\sum_n \frac{1}{2} \cdot (|RL_i|) \cdot (|RL_i| + 1)\right) + \sum_{\forall p \in P} \sum_{\forall s \in S} \frac{1}{2} \cdot (|T_{p,s}|)\,(|T_{p,s}| + 1)$$

where $T_{p,s}$ is the set of transitions in process p that have a source state s

The next step is the identification of solid connected components in the TRG to create the Minimal Candidate Schedules (MCS). This process occurs using the wave-front algorithm described in Chapter 5. In brief, the algorithm can be summarized as follows:

1. Divide the graph into solid connected components

2. Begin at any Transition Edge vertex.

3. Add all solid connected unvisited Rendezvous vertices to the path
4. For each newly added Rendezvous vertex, compute a set of next Transition Edge vertices such that there are no mutual exclusion edges between them and vertices already in the path
5. Repeat until all reachable Transition Edge vertices are seen
6. Repeat for all mutual exclusive neighbors of the original starting node
7. Repeat for all disjoint components

We will attempt to prove the complexity bound on this algorithm in terms of the TRG graph. Let us begin by considering some edge cases. If the original structure is a long chain of conjunctively composed rendezvous, i.e. if each process has one state, with one transition edge, and the whole system contains n processes, and $n-1$ rendezvous, then we can say that the running time for the algorithm is $O(n)$. Now consider a case where we have a system with only 2 processes, but n rendezvous communicating between them such that the n rendezvous are nondeterministically mutually exclusive. Such a case may be easily shown to be analogous to the situation where there are n transition edges that use the same rendezvous to communicate between the 2 processes. To be more specific, we are considering a biparty rendezvous which has n "+" role labels and 1 "-" role labels, and thus has a total of n communication patterns. In this situation, the running time of the algorithm is $O(n)$. If we consider a case where we have a single biparty rendezvous communicating between 2 processes with n "+" role labeled nondeterministic mutually exclusive transition edges in one process and a similar n "-" role labels in the next process, i.e. the rendezvous has n^2 ways of occurring, then the running time of this algorithm is $O(n^2)$.

More generally, the running time of this algorithm can be defined using the following recurrence relation

$$T(n) = \begin{cases} \Theta(1)\, n = 1 \\ T(n-1) + deg(n)\, n\ is\ odd \\ \frac{V_{valid}}{d}^{d} (T(n-d) + d)\, n\ is\ even \end{cases}$$

While the general solution to this may seem to be exponential, in practice, it is much more restricted. The presence of Mutual Exclusion both deterministic and static can be used to remove complexity at each step in the wavefront computation. As a result, it is not simply sufficient to examine the entire state space. The presented recurrence relationship represents a loose upper bound on the running time, as any tighter bound will necessarily depend explicitly on the input design.

An alternate way to examine this is that in the worst case, the number of MCS is bounded by the powerset (i.e. set of all sets) of the number of transition vertices in the TRG. As a simple pruning mechanism, sets that contain just a single element will be disregarded, as they cannot form valid schedules. We can also disregard sets that may contain mutual exclusion. As a simple number, this is equal to the number of mutually exclusive edges in the TRG. This loose bound is in practice further reduced by the number of roles that each rendezvous has, a factor that is not expressed here. This can be expressed as :

$$|MCS| \leq 2^{\sum_{\forall p \in P} |T_p|} - \left(|E_{TRG}| - \sum_{\forall r \in R} \left(\sum_n |RL_i|\right)\right) - \sum_{\forall p \in P} |T_p|$$

After the generation of the minimal candidate schedules(MCS), these can be related by pairwise examination of the TRG for nondeterministic relationships. By applying these relationships, we can create the Occurrence Relationship Graph, defined as $G_{ORG}(V_{ORG}, E_{ORG})$. From the construction algorithm, we know that V_{ORG} is the number of MCS generated by the wavefront algorithm. As a very loose upper bound, we know that $|V_{ORG}| = |MCS|$, or that the total number of candidate

schedules must exist within the power of the transition vertices in the TRG. As these come from the description of the system, we can say that the number of vertices in the ORG is a power set of the union set of all transitions. We can give a very loose bound on the number of edges in the ORG as the square of the number of vertices, i.e. the case where the ORG is a fully connected graph. In more practical terms, each edge in the ORG is caused by a Nondeterministic Mutual Exclusion edge in the TRG. In turn, each mutual exclusion edge in the TRG is caused by an edge in the original system description. In order to create a tighter upper bound, let us consider the sources of nondeterminism separately.

For nondeterminism arising from variability, i.e. from cases where two transitions are sharing the same rendezvous role label, we can compute its contribution to the edge count as the worst case, where the role labels for the rendezvous create a situation where the number of ORG nodes is in the product space of the roles, and the resulting ORG is fully connected.

$$E_{ORG,v} = \sum_{\forall r \in R} \prod_{\forall rl \in RL} (|rl|^2)$$

Similarly, we can consider the contribution of disjunction, i.e. the cases where two transitions share the same source state. In this situation, we can compute the number as the product space of the number of transitions that share the same source state

$$E_{ORG,d} = \sum_{\forall p \in P} \sum_{\forall s \in S} \prod_{\forall t \in T_{p,s}} |V_{ORG,t}|$$

where

$V_{ORG,t}$ is the set of vertices in the ORG that contain the transition t

In a similar fashion, the number of edges arising from race conditions on shared variables can be found as:

$$E_{ORG,r} = \sum_{\forall p \in P} \sum_{\forall v \in V} |V_{ORG,v}|$$

where

$V_{ORG,v}$ is the set of vertices in the ORG that write to variable v.

The total number of edges in the ORG, thus can be given as :

$$E_{ORG} \leq E_{ORG,d} + E_{ORG,t} + E_{ORG,v}$$

The reason this might be less is clear. The case for the variability based non determinism assumes the worst possible case. Furthermore, some of the edges may be double counted, as the graph is not a multigraph, and thus multiple mutual exclusion edges from the TRG may result in only one edge in the ORG. Finally, the ORG does not include deterministic mutual exclusion, i.e. edges where the guard conditions provably cannot overlap. As a result of these factors, the number of edges in the ORG will be much lower than the bound in practical designs.

6.4 Scheduler Description

As we have discussed in Chapter 4, the scheduler is a very important part of what makes Lyra unique among rendezvous based approaches. In this section, we will examine the scheduler's construction in formal terms, and try to express the scheduler using a formal notation. Finally, we will demonstrate that the constraints on the formal model of the scheduler can be translated back into the original model and verify that the provision of nondeterminism is a useful feature.

Firstly, we need to define the notion of a Finite Automaton (FA) as the following 5 tuple [Carrol and Long, 1989], [Hopcroft et al., 1979]:

$$A = \langle I, S, \delta, S_0, A \rangle$$

where:

I is the input alphabet

S is the set of states

$\delta : S \times I \rightarrow 2^S$ is the next state function

S_0 is the initial state marking

A is the set of accepting or final states

A *run* of a automaton is defined as a ordered set of states that the automaton progressively passes through. More formally, a run using the input string x (denoted as s^x) is defined as :

$$s^x = s_0^x, \ldots, s_n^x \text{ where } s_0 \in S_0 \text{ and } s_i \in \delta\left(s_{i-1}, x\right) \subseteq S \text{ and } x = (x_0, \ldots, x_{n-1}) \text{ where } x_i \in I \; \forall \; 0 \leq i < n$$

A basic property of an FA is the accepting property, defined as follows :

$$\text{The string } x \text{ is accepted if and only if there exists a run } s^x \text{ such that the final state } s_n^x \in A$$

For the sake of convenience, we can extend the δ function defined so that it maps strings. i.e. $\delta : S \times I^n \rightarrow S$. Thus when we designate $\delta\left(s, x\right)$ this returns the final state after the string x has been applied to the automaton.

Further, we define as the *language* of a FA as the set of all accepted strings that begin from the initial state and finish in the accepting state, denoted as L.

More informally, we can say that an finite automaton will accept a given series of inputs only if the final state of the automaton after applying the inputs is in the accepting set. Thus, by clearly defining the accepting set, we can define which strings are allowed and which are not.

This discussion is general for both deterministic and nondeterministic systems. The main difference between a deterministic finite automaton (DFA) and a nondeterministic finite automaton (NFA) is in the δ function. In the case of a DFA, there is only one possible next state, given a current state and input. In the case of an NFA, there are multiple next states possible for a given present state and input.

If we consider the fundamental problem of scheduling in Lyra, we can see that it is analogous to generating a NFA where the input alphabet is a set of schedules, and strings are a series of schedules that are going to be attempted to be run in parallel. If the string is accepted, then the scheduler is signaling that this set of schedules can be executed concurrently for this iteration. The nondeterminism from the original description causes a choice in schedules at runtime that is resolved by the application of a policy. This policy application can be seen as the construction of an equivalent DFA from the scheduler's NFA, while keeping in mind the criteria to be optimized.

Thus, the formal Lyra scheduler can be expressed as the n tuple

$$\langle M, S, \delta, S_0, A \rangle$$

where:

M is the set of input Schedules

A is the set of acceptable Schedules

As we can see, based on the ORG, we can compute A as $A \in 2^{|V_{ORG}}$, or that the accepting set is the set of all schedules that can be composed out of the minimal candidate schedules such that no subset violates the mutual exclusion rule. Consequently, the language of this machine is the power set of minimal schedules.

Throughout the design of the Lyra methodology, a lot of emphasis has been placed on the use of nondeterminism. Let us examine some of the practical aspects of the use of nondeterminism. An important factor is that by construction, the ORG captures the nondeterminism in the original description. The presence of non determinism in the original description is the necessary and sufficient condition for the presence of the same in the scheduler.

The ORG as presented contains concurrency between unrelated MCS. Between related MCS, we have explicit nondeterminism. Thus, as shown in [Drusinsky and Harel, 1994], the ORG definition follows the E [Kozen, 1976] and A

[Chandra and Stockmeyer, 1976] [Chandra et al., 1981] properties. Thus, the inclusion of AND and OR semantics in the ORG make it double exponentially more compact than the original description. For the ORG based scheduler, the AND semantics are implicit in the concurrency assumption of unrelated MCS. The OR semantics are due to the mutual exclusion edges between the various processes.

In addition, a similar analogy can be drawn from the original description. As Lyra permits the use of conjunction and disjunction, the original description posses AND and OR semantics. The lack of a clock imply that each process is independent, outside of rendezvous based synchronization. Thus the A property from [Drusinsky and Harel, 1994] applies. The mutual exclusion based form of nondeterminism in the rendezvous selection, modeled as variability and disjunction, satisfies the E property. Thus, the presence of nondeterminism in Lyra allows the input size, i.e. the description of the system, to be exponentially smaller than it would have had to be in the absence of nondeterminism.

The major problem with the ORG based scheduler as a NFA is that a NFA cannot be synthesized. While the NFA to DFA construction is well known [Hopcroft et al., 1979] [Carrol and Long, 1989], it is polynomial in the size of the DFA. As the scheduler must be implemented in hardware, this polynomial size becomes a bottleneck. In order to allow some designer input, and constrain the size of the translation, we introduced the notion of a scheduler policy. While informally, the policy is simply a way for the scheduler to make deterministic choices where nondeterministic ones exist, it is formally a construction of an equivalent DFA, based on the formulated scheduler NFA, such that the heuristic constraint is held.

The GWO scheduling policy, discussed in 5.5.1 is an optimal policy, and as such follows a traditional NFA to DFA expansion. The primary differences are due to the constraint on the input alphabet (as the input alphabet will always be a strict subset

of the total schedule space), and the usage of rendezvous weight allow the creation of a tree. This tree, while not always strictly polynomial, presents a definite overhead, as intermediate nodes in the tree must perform and store computation. More formally, the GWO constraint is a simple modification to the DFA construction, where the input alphabet is a subset of the working alphabet. In other words, the internal nodes of the tree correspond to intermediate states introduced by the conversion of the NFA into a DFA. Each transition is clearly disambiguated and thus becomes resolvable.

The SLWO scheduling policy, discussed in 5.5.2 is a much easier to implement policy. By sacrificing the global nature of the weight assumption, the SLWO policy is able to restrict the overhead due to the scheduler. In construction terms, it is based on the notion of weight, and the reduction in the state space by strictly ordering the possible transitions. In the case of nondeterminism, a transition with a lower weight is simply removed. Thus, the DFA will contain exactly as many states as the NFA, but with a reduction in the number of transitions. The equivalence of this DFA to the original NFA is one way. In other words, for a fixed set of inputs, the DFA from the application of the SLWO policy will behave like one possible run of the original NFA. An alternate way to consider this is that the DFA is constructed by simply examining the set of transitions and discarding any duplicate transition.

6.5 Implementation

One of the main issues that remains, as might be obvious to the reader is the fact that a FA is usually implemented as a sequential machine, whereas in this case, we need a combinational implementation. However, this is not a hurdle here. We know that by the expansion of the entire state space of the storage element of a sequential machine, we can implement it as a combinational circuit. In this case, there are no

internal variables, except for the state. If we examine the NFA, we can see that the internal states actually correspond to sets of MCS that have been grouped together. Thus, as long as the combinational circuit accounts for the possibility of considering multiple MCS in the final schedule, there can be a combinational implementation. The tree like implementation in the GWO case, and the separated directed acyclic graph implementation in the case of SLWO policy actually ensure that the computation of sets of MCS happens in parallel. In addition, since the SLWO policy is only one way equivalent, some of the sets of MCS are actually ignored.

Chapter 7

Lyra Tool Flow

In this chapter, we describe the implementation of the Lyra methodology. We describe the simulator and synthesis tools that were implemented. We provide a description of the hardware synthesis tool, including how the graph based approach is mapped into hardware.

7.1 Overview of the Lyra tool flow

The Lyra methodology has been implemented as a set of software tools - one for synthesis and another for simulation. These tools are written in C++ and use the Standard Template Library(STL) and the Boost C++ Libraries for the implementation. Due to the way the methodology is constructed, these tools are able to share a large part of their codebase. Thus, the tools access a shared library that provides the front end to the language parser, its abstract syntax tree and intermediate representation generator, and finally the ORG generator. The common ORG is then used differently by the simulator and the synthesis tool to dynamically execute the system and generate a SystemC representation of the system respectively.

7.2 Simulator

The simulator for Lyra based systems is a straightforward implementation based on the ORG. The ORG is presented as a graph to the simulator. At this point, the synthesis policy is mapped onto the graph. In the case of GWO policy, the structures are created to store the system in its tree based representation. In the case of the SLWO policy, this changes the graph into a directed acyclic graph. In the case of the software implementation, this requires simply making minor changes on the direction of the given edges.

The structure initialization is performed by analyzing the model. Once this phase is complete, the simulator presents an interactive shell where it can be run, stepped and variables and data can be examined. At each run time iteration, all the ORG vertices are evaluated to determine if their guard condition and state based conditions are satisfied. The valid MCS are then marked in the internal policy dependent representations to determine the set of MCS that can occur. These MCS are then scheduled to occur and their semantic actions are evaluated. Finally, the variables are updated from the semantic actions and the system proceeds to the next state.

7.3 Synthesis

In order for the modeling methodology to be able to model hardware, it would be useful to examine how synthesis occurs. We can define this problem simply as follows : given a concurrent system described using the proposed methodology, create a synthesized version that can be implemented in hardware, using existing synthesis flows.

The following section describes the synthesis flow in further detail. In brief, the ORG based approach has given us some initial insight into this problem. As the

ORG will express all possible candidate schedules in the system, it becomes possible to create hardware equivalents for each of them. The problem of selection becomes simpler as we can test all these possibilities in parallel. Furthermore, we can observe that each ORG vertex, or minimal schedule, is self contained and therefore can be implemented as a single block. The main problem of synthesis can then be distilled into the creation of the connections between ORG blocks based on the scheduling policy. Another area of concern, especially for the practical use of this methodology, is in the estimation of the performance of the generated hardware and identification and optimization at a high level of the system to improve such performance. Ideas for the same are discussed further in this section.

7.3.1 Overview of Approach to Hardware Synthesis

This work proposes a general synthesis flow that is shown in Figure 7·1. The basic flow accepts a process network as input to the synthesis tool. The tool creates output files containing circuit descriptions, which are then synthesized to hardware using an existing tool flow. These output files are behavioral descriptions in SystemC for integration with SystemC based hardware synthesis tools.

We can examine the basic tasks the synthesis flow must perform in Figure 7·2. The synthesis flow is conceptually divided into two sections - a target language independent section and a target language dependent section. This is to allow the synthesis framework to support multiple target languages. The input process network is used to generate its ORG. Then, each MCS, register, and datapath are translated into synthesis friendly descriptions. These translations basically involve ensuring naming uniqueness, creation of modules and hierarchy, handling of datatypes, and reordering of statements as appropriate. Finally, the whole system is translated into a synthesizable, language independent structure. At this point, language specific routines are

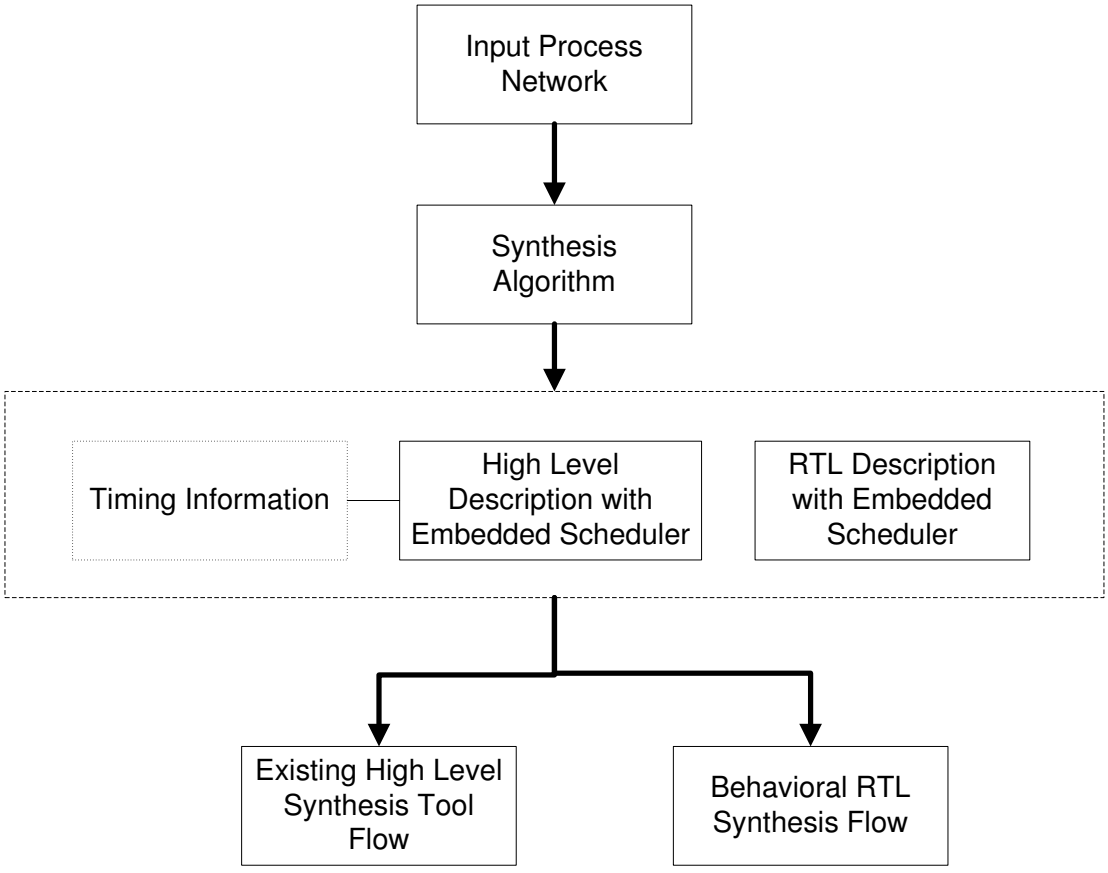

Figure 7·1: High Level Overview of Synthesis Flow

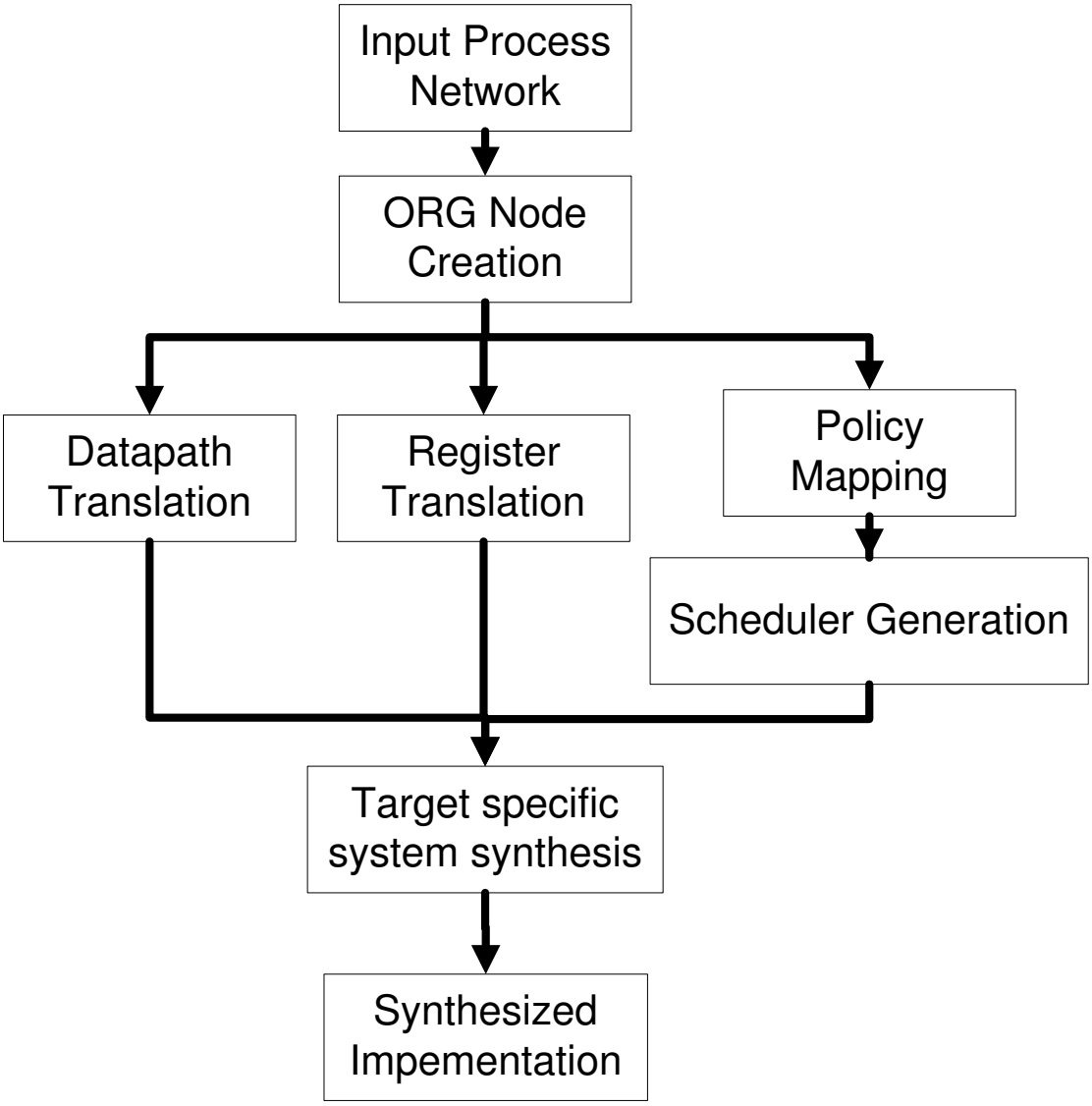

Figure 7·2: Detailed view of Synthesis Flow

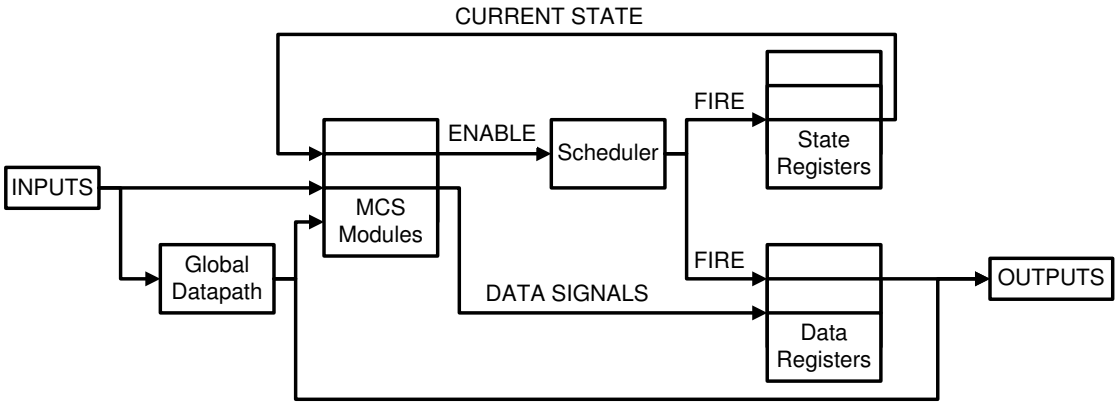

Figure 7·3: A general synthesized system

used to create the final output in the specified language (such as SystemC) from this structure.

In general, a Lyra system is synthesized into a description shown in Figure 7·3. Each MCS is created as a separate block of hardware. The enable signals from the MCS blocks are sent to the scheduler. The scheduler accepts the enable signals and produces a set of fire signals, one for each MCS block. These fire signals are sent to the state registers and the data registers. At the state register, receiving a fire signal performs an appropriate update of state. As the fire signals correspond to the MCS, the knowledge of the fire signal being active is sufficient to compute the next state. For the data registers, the data computed in the MCS is sent to the data register and is selectively multiplexed into the register, based on the fire signal. The current state is fed into the MCS blocks, where it is used to generate the MCS ready signal. The data register values are fed back into the MCS or can be sent to the toplevel output.

7.3.2 ORG Node Creation

Each ORG node in the software scheduler is synthesized into a unique combinational block. This block contains two major components. One portion of the block is the data flow associated with the MCS. Further, conditional guard statements on

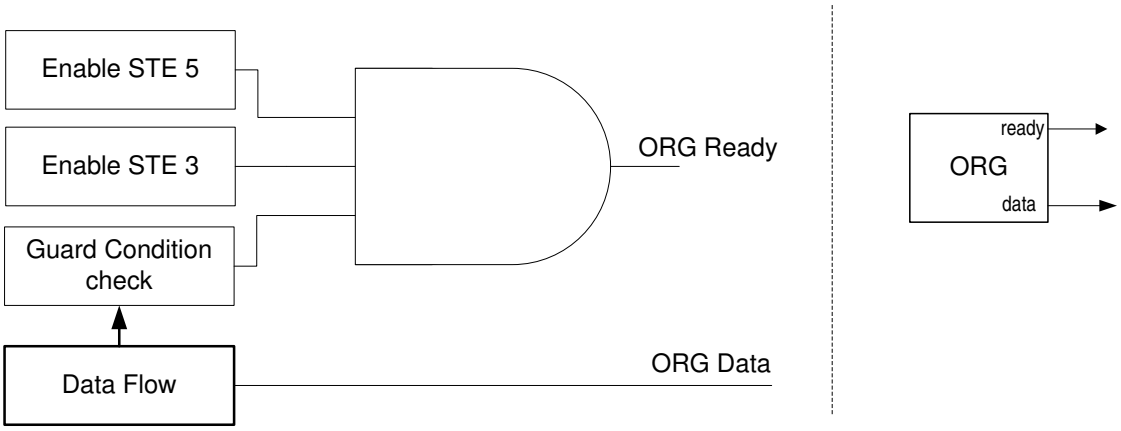

Figure 7·4: Synthesis of ORG Nodes

each transition edge would be synthesized to an appropriate combinational hardware equivalent. The other portion of the logic computes the enable signal for the MCS by examining the values of the state registers associated with the transition edges, and contains inputs from the synthesized guard condition hardware.

In Figure 7·4 is an example of a synthesized version of an ORG block with one ready output and one data output. The readiness is based on whether each transition edge involved is incident to the current state of its owner process, and whether the guard conditions on all transition edges are true. When all are true, ready becomes 1. The dataflow associated with the 2 edges, specifically the register writes are sent to the data output.

Note that in a logic block in Figure 7·4, evaluating the guard condition of a transition edge may depend on the data value brought to the edge via the synchronous data flow through several conjunctively composed rendezvous. Since all conjunctively composed rendezvous must appear in the corresponding ORG node and thus the logic block, the evaluation of the condition is self-contained in the logic block. The actual data flow block is synthesized during process synthesis using existing behavioral synthesis tools.

7.3.3 Policy Mapping

A major area of the work of the scheduler and, consequently, the synthesis algorithm, is in the implementation of a specified scheduling policy. The exact synthesis algorithm for this stage depends on the policy being implemented. In general, due to the design of the synthesized system, the specifics of the scheduler's implementation do not matter. The scheduler is simple a block that has 1 bit enable signals for each of the MCS blocks, and produces a 1 bit fire signal for each MCS. The fire signal corresponds to whether that particular MCS will be active in that iteration. Below are examples of synthesis for the policies previously discussed in Section 4.3.

Static Local Weight Optimal (SLWO) Policy Synthesis

When synthesized into synchronous hardware, the behavioral specification in Figure 5·8 will become Figure 7·5. Each rectangle in Figure 7·5 is a block of combinational logic derived from one ORG vertex. Based on the priority-based scheduling scheme, the ready and enable signals can be chained according to the previously mentioned DAG.

This simple policy is the currently implemented synthesis policy. This is because this policy exhibits significantly less overhead and is far more amenable to hardware implementation. The policy itself is not very far away from optimal, and does allow for a lot of concurrency in many practical cases.

7.3.4 Other considerations for Synthesis

The synthesis tool is implemented using a framework that allows for the extension of the tool into different flows. While the current target is to SystemC, it would be possible to generate RTL verilog from the the synthesis tool. In addition, in SystemC, modules communicate via sc_signals that behave like Verilog registers, i.e. follow non

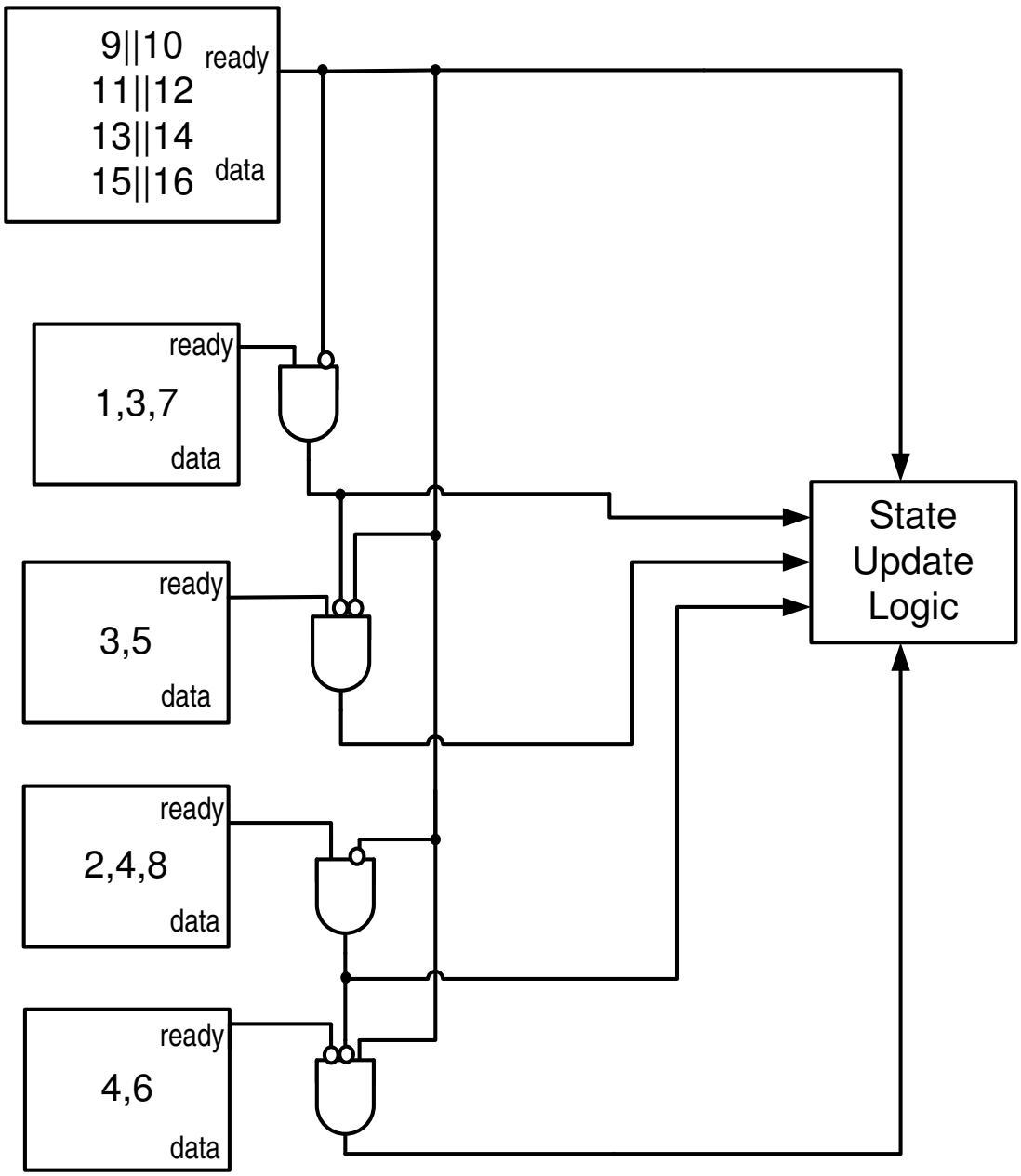

Figure 7·5: Synthesized scheduler using SLWO policy

blocking or delayed update semantics. In order to bypass problems of synthesis caused by this, the SystemC model is forced to make progress by making each method have a zero time wait function, that allows for a delta update. This is a practical method to model combinational wires in SystemC.

Chapter 8

Comparison to Other Approaches

In this chapter, we compare Lyra to other rendezvous based approaches. We use the graph based analysis that we have developed in this dissertation to provide a common framework for comparison. We begin by explaining the effect of each feature of the rendezvous on the graphical method in general, and then considering the specific languages previously presented.

8.1 Effects of Rendezvous Features

Using the graphical tools presented in Section 5, we can compare different rendezvous based modeling methodologies. The complexity of the candidate schedules is a parameter that depends heavily on the construction of the system and the features of rendezvous being used. In the simplest case, we consider a model where rendezvous are neither variable, nor conjunctively or disjunctively composed. In this scenario, every solid-edge connected component in the TRG contains only two transition vertices and one rendezvous vertex. Each connected component forms a minimal candidate schedule. Thus the number of candidate schedules will always be exactly equal to the number of rendezvous in the system.

The introduction of variability alone will increase the number of ways a particular rendezvous can occur, and hence requires us to examine the number of labels of a role of a given rendezvous that exist. Thus it increases the number of candidate schedules.

As previously mentioned, a variable k-party rendezvous with ni processes for each role can be decomposed into ni bi-party non-variable rendezvous. Since each non-variable rendezvous maps to one candidate schedule, the variable rendezvous will result in ni candidate schedules.

Disjunctive composition alone will not create any effects on the number of candidate schedules; however, it may affect the run-time choice of candidate schedules since it may introduce NME edges into the TRG and ORG.

Conjunctive composition alone, on the other hand, increases the sizes of the solid-edge connected components in the TRG by bonding rendezvous together into multi-party candidate schedules. Each component will include more than one rendezvous vertices, but still corresponds to only one candidate schedule.

However, when combined together, these features collectively increase the number of candidate schedules. For example, given two variable bi-party rendezvous, one with n processes for the '+' role and one for the '-' role, and another with m processes for its '+' role and one for the '-' role. When used independently, the first one introduces n candidate schedules and the second introduces m. However, when the two rendezvous are bonded together by conjunctive usage of their '-' roles, their number of the candidate schedules multiplies to n*m. In the example TRGs presented in Figure 8·1, the rendezvous a has three '+' roles and one '-' role, and the rendezvous b has four '+' roles and one '-' role. When there is no conjunction, this system has 7 candidate schedules. However, when the two '-' roles are conjunctively composed, the number of candidate schedules increases to 12.

The free combination of all of these features, particularly the increase in possible candidate schedules from variability and the implied dependence due to conjunction, makes the analytic determination of number of candidate schedules hard.

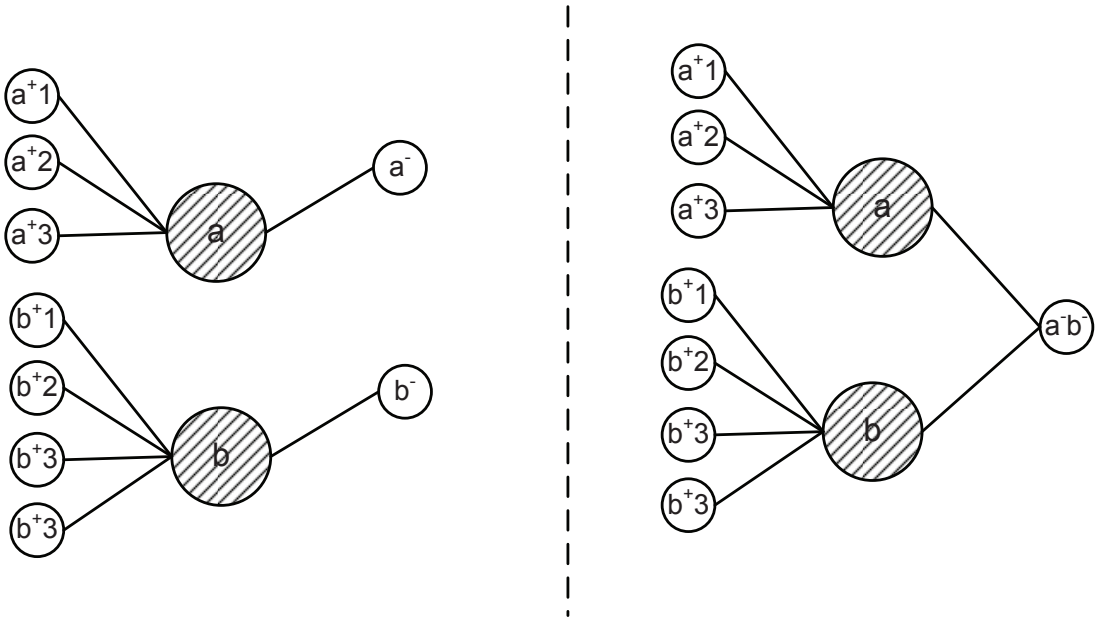

Figure 8·1: Complexity Increase due to conjunction

8.2 Graphical Analysis

The TRG presents us with a simple graphical representation of the communication complexity of the system. Each unique solid connected component in the TRG, or an ORG vertex, corresponds to a communication pattern in the system. The NME edges provide us with Transition Edges where global scheduling is necessary. In the case of the languages described in Chapter 2, the effects of the restrictions on the TRG and ORG are quite clear.

In the case of SHIM, the TRG will be free of NME edges. Each connected component contains one rendezvous vertex and two transition vertices (or more for multiparty barrier). It directly maps to a disconnected vertex in ORG.

In the TRG for an Ada model, each connected component contains exactly one rendezvous node, one "callee" transition node, and n (n≥1) mutually exclusive "caller" transition nodes. The component will be mapped to n mutually exclusive nodes in

ORG.

In the TRG of an Occam model, each connected component by solid edges contains one rendezvous vertex and two transition vertices. Several transition vertices may be connected by NME edges due to disjunction within a process. Due to the lack of conjunction and variability, each TRG NME edge directly corresponds to exactly one ME edge in the ORG. Thus, in the ORG, there are exactly as many ME edges as NME edges in the TRG. The Haste TRG and ORG should be similar to that of Lyra's. However, the main difference is that Haste assumes that all disjunctive compositions are resolved by each process locally. Similarly, variability in Haste is always arbitrarily resolved. In graphical terms, Haste treats all state transition edges as DME related, and thus the ORG contains disconnected vertices. When implemented in hardware, such a Haste system would be nondeterministic. In a software simulation, such a description should deadlock. Bluespec based models look exactly like Lyra models, but instead of resolving nondeterminism, they sforcibly remove it by composition of rules. Thus, the job of creating the scheduler is handed to the designer.

Chapter 9

An empirical example of a Lyra based system

In this chapter, we describe the practical use of Lyra in modeling a heteregenous system on chip. The system consists of a cycle accurate processor model that interacts with a transaction level model of a PCI Express bus network created using Lyra. We describe the overall system and then detail the PCI Express topology and its implementation in Lyra.

9.1 Introduction

The expressive power of Lyra lends it to the easy modeling of a wide variety of systems. A useful way to model Systems-on-Chip is by performing heterogeneous simulation, where different components of a system are at different levels of abstraction, allowing the user to balance simulation speed with accuracy in a finer grained manner. In this case study [Venkataraman et al., 2009a], we will examine a cycle accurate simulation of a PowerPC like processor communicating with a transaction level model of a simple PCI Express Network [Automation, Design and Committee, Standards, 2006]. The PCI Express network is described using Lyra contains a simple root complex, switch and two endpoints. The processor model is a 5 stage in order pipelined processor that supports a PowerPC instruction set. These models are integrated into a system

simulator described using SystemC. The following sections examine each component of the system model in further detail.

9.2 PCI Express Model

Lyra PCI Express Model In the following section we first examine the PCI Express protocol [Budruk et al., 2003] in general and then the implementation of some of the features using Lyra to create the transaction level model. After a general discussion, we delve into the modeling of each component of the protocol in detail, further elaborating the need for certain implementation decisions based on the protocol.

9.2.1 PCI Express Overview

PCI Express is a refinement of the previous Peripheral Component Interconnect bus architecture. PCI Express is a high speed, scalable, packet based, point to point bus protocol. PCI Express networks usually contain at least one or more root complexes, switches and endpoints that are connected by PCI Express links. A processor or set of processors is connected to each root complex and can issue commands into the network. The topology is logically directed from the root complex to the endpoints, where the direction towards the root complex is called upstream and that towards the endpoints is called downstream. The PCI Express protocol defines 3 types of packets for the 3 layers of the OSI model in which the protocol is defined. Transaction Layer Packets (TLPs) for the transaction layer, Data Link Layer Packets (DLLPs) for the data link layer and Physical Layer Packets for the physical layer. From a high level perspective, the main task of a PCI Express network is to transfer data in the form of TLPs from one component to another. The data link and physical layer packets are internal packets that are not directly controlled by the system designer.

The physical layer level components consist of the electrical components necessary

for bus communication as well as low level function such as device discovery, packet framing and link training and initialization. Data link layer elements include link functionality elements such as components for retransmission of unreceived/corrupted data, acknowledgment of packets, flow control and packet CRC generation and validation. The transaction layer is mainly in charge of reassembling transactions based on packets received, replying to transaction requests, accepting new packets from the software level and allowing configuration of the device.

9.2.2 Overview of the Lyra implementation

In this case study implementation of PCI Express, we will model mainly the transaction and data link layers of the PCI Express protocol as modeling the physical layer would be orthogonal to our problem. We will be considering a simple tree like network, consisting of one root complex, one switch and two endpoints, where the links between components are rendezvous. This is shown in Figure 9·1. The small squares within each PCI Express component are the ports the link connects to. Upstream ports are colored white and downstream ports are shaded gray. A subset of features of the PCI Express will be supported at first, with possible extensions for other features later on.

The model was developed while keeping in mind the principles of code reuse and hierarchical design. As a result, it was noted early on that the primary difference in functionality between the different PCI Express components was in the transaction level behaviors. A common module, the PCI Express Port, was created to handle the data link layer functionality of a PCI Express component. However, as the behavior is not quite identical, this module is controllable via parameters to modify the behavior depending on whether it is instantiated in a PCI Express switch, endpoint or root complex. In the remainder of the section, we examine each of the basic Lyra modules

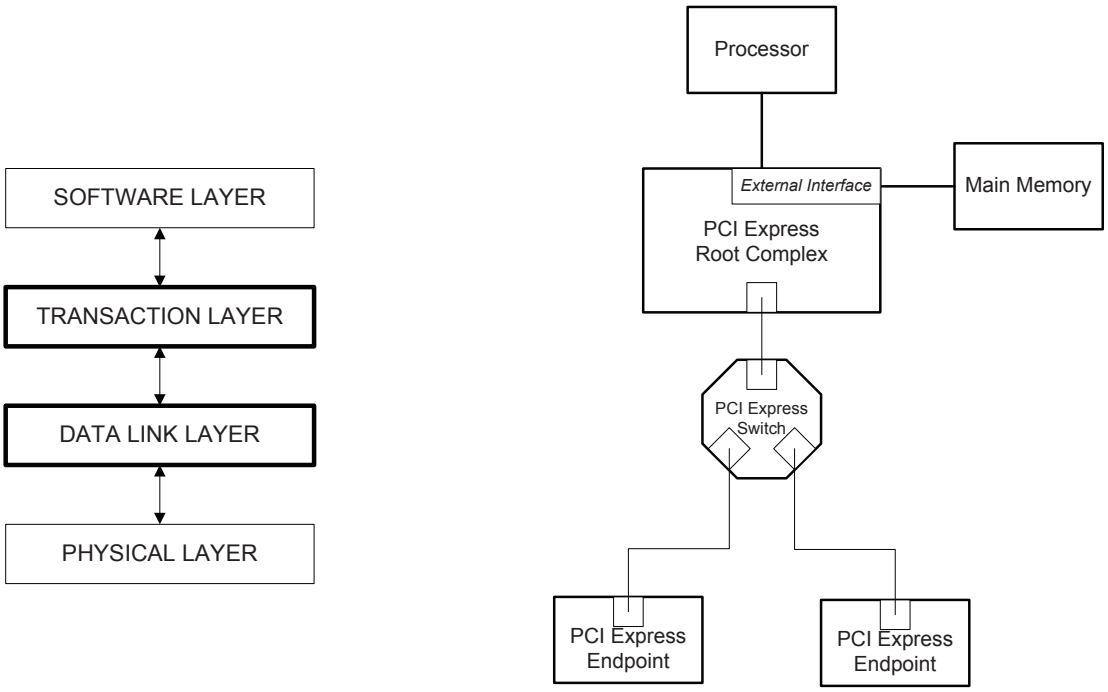

Figure 9·1: PCI Express Network Topology

that are created, the protocol basis for each and a little bit about the implementation of the module behavior in Lyra. As a quick reminder, the Port module is not part of the protocol specification, whereas the root complex, switch and endpoints are specified components of PCI Express.

9.2.3 Port Module

The Port Module abstracts the data link layer behavior for all PCI Express components. It implements two rendezvous for the public input and output portions of the PCI Express component and two additional rendezvous for the private transaction level input and outputs. The terms public and private are with respect to the scope of the parent component as part of which the port has been instantiated. As such, the port contains the data link layer functionality of the PCI Express protocol. While not part of the PCI Express specification, creation of the port module allows

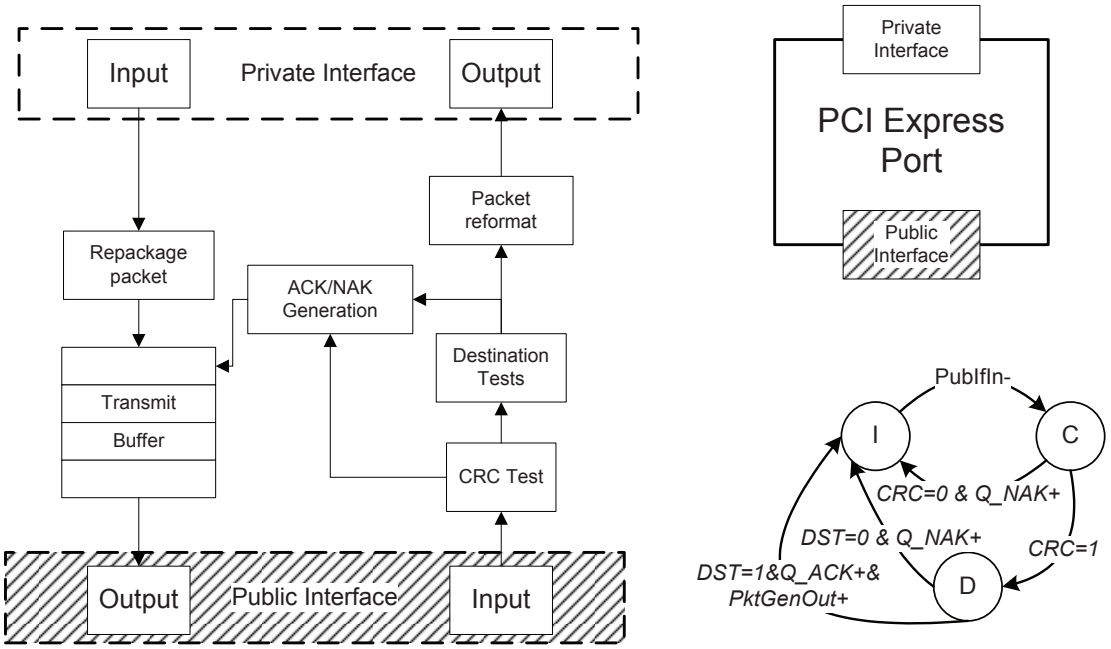

Figure 9·2: PCI Express Port Model

for simplification of the design and code reuse.

Shown in Figure 9·2 is a simplified state diagram of part of the Lyra implementation of the port. The CRC test process accepts input from the rendezvous connected to the public interface input. This causes the machine to transition to state C where it computes the CRC and checks it. If the CRC if the packet is valid, it proceeds to state D where the destination of packet and the addressable range of the port are tested. If the test is passed, it queues an acknowledgment packet for transmission via the Q_ACK rendezvous, while passing the packet to the next processing step by the PktGenOut rendezvous. If the destination test is failed or the CRC test is failed, a negative acknowledgment is queued through the Q_NAK rendezvous. After any of these paths, the process returns to the initial state and prepares for the next input.

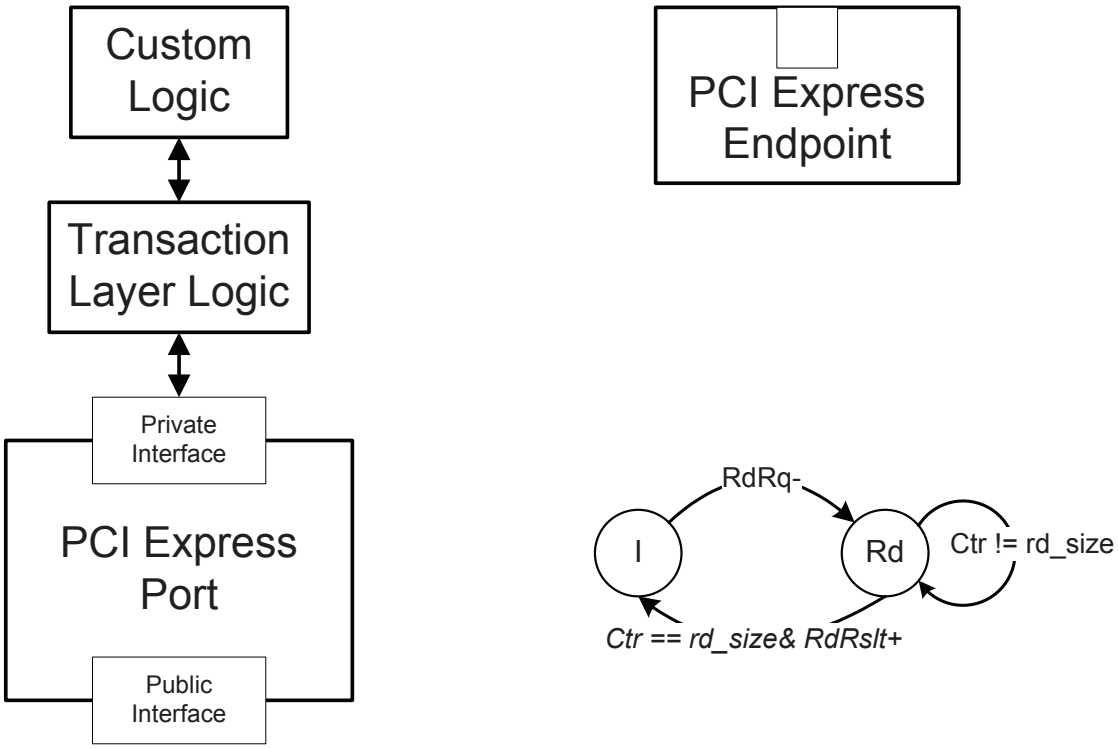

Figure 9·3: PCI Express Endpoint

9.2.4 PCI Express Endpoint

A PCI Express endpoint is shown in Figure 9·3. It consists of a port and transaction layer logic for the protocol. In addition, there exists some custom software layer logic that implements the actual behavior of the endpoint. In this case study, the behavior is to present a small byte addressable memory that can be read from and written to. The main purpose of the transaction layer logic is to test if a packet is destined for this endpoint and to ensure that commands to be sent to the custom logic are valid ones. The custom logic implements a simple read write interface that reads from and writes to the memory.

Figure 9·3 is a state diagram for the process that reads from the memory. New requests arrive in the RdRq rendezvous, with the base address and a counter for the

number of bytes to read. A counter is initialized and the process moves to the Rd state where it loops reading 1 byte at a time into a register. When the full number of bytes is read, the process outputs the data on the RdRslt rendezvous and returns to the initial state I.

9.2.5 PCI Express Switch

A PCI Express switch has multiple sets of public interfaces, each set containing an input and output, with exactly one upstream interface and multiple downstream interfaces. Packets arriving at any of the public interfaces are routed based on the type of addressing used, the expected destination of the packet and the direction the packet is traveling.

In Figure 9·4, the state diagram for the process watching the input on the upstream interface is shown. Upon receiving a packet on the UpIn rendezvous, the process transitions to state D. In this state, the packet's routing information is checked against the known data for the two downstream ports and bits D1 and D2 are set if the packet is bound for the first downstream port or the other. If the packet is bound for a port, then the process queues it up with that port's output. If there is no match, an error message is queued up to be returned to the upstream link.

9.2.6 PCI Express Root Complex

The main role of a root complex is to act as an intermediary between the processing element, main memory and the PCI Express network. In addition to this role, the root complex also behaves as an endpoint, and in the case of a root complex with multiple downstream ports, as a switch.

The root complex implemented in Lyra for this case study, shown in Figure 9·5, possesses only one downstream port and additional transaction level queuing logic.

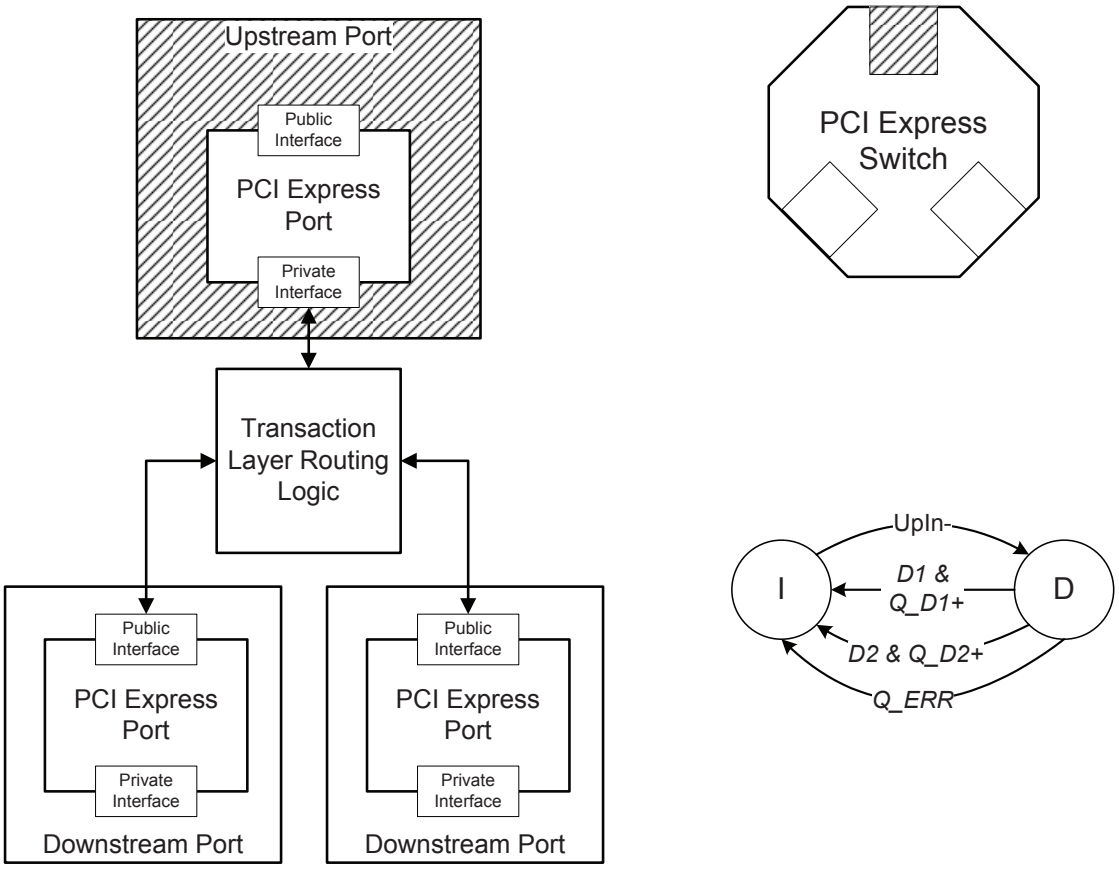

Figure 9·4: PCI Express Switch

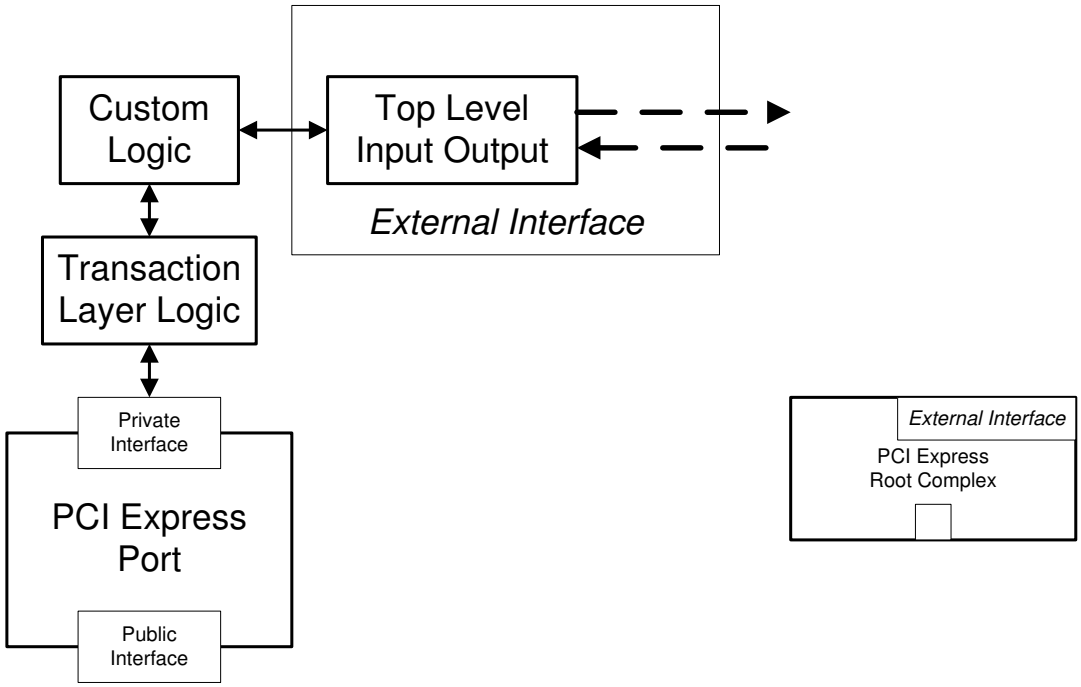

Figure 9·5: PCI Express Root Complex

The queuing logic is connected to a pair of top level, signal based, input and outputs which are fed sequential data by the CPU. This is discussed in greater detail in section 9.4.

9.3 ADL Model

Architecture Description Language (ADL) and micro Architecture Description Language (uADL) are two open source tools developed at Freescale Semiconductor, that allow the quick and efficient exploration of the design space of a new microprocessor or the functional modeling of an existing one. By describing the architecture of a microprocessor using ADL and its micro-architecture in uADL it becomes possible to create a cycle accurate instruction set simulator of a processor. The normal design flow using ADL is shown in Figure 9·6. ADL and uADL have a custom language that

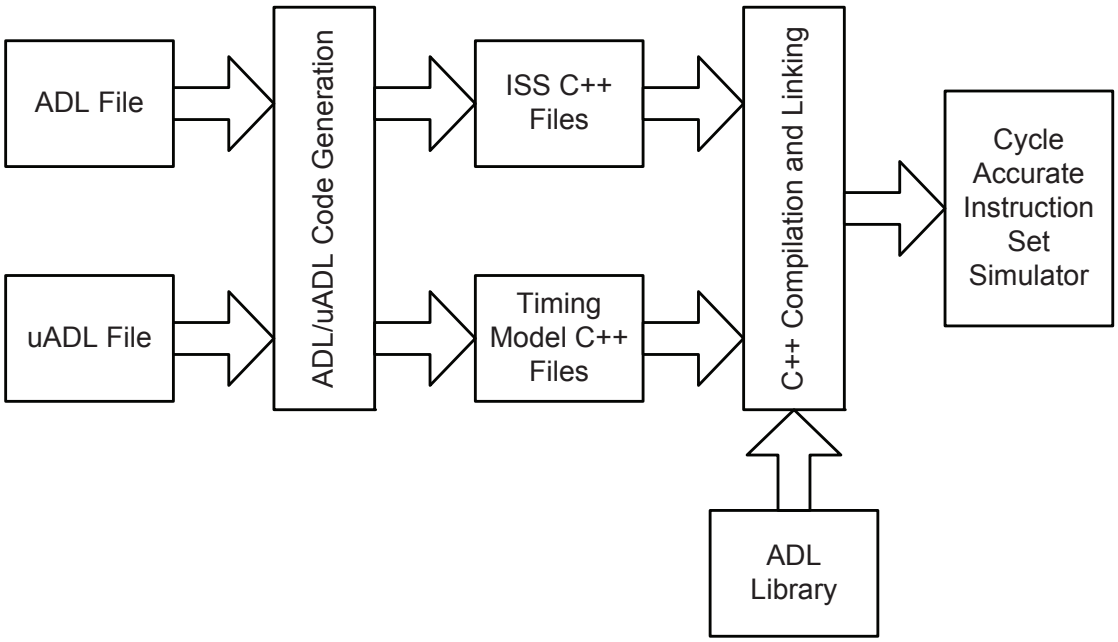

Figure 9·6: ADL Design Flow

enables the description of a microprocessor core. These two files are run through ADL tools that generate a pair of C++ header and source files that describe the behavior of the given processor. These files are compiled and linked against the uADL libraries that provide the external interface, instantiation, and library components necessary to produce an executable instruction set simulator.

The generation of the executable instruction set simulator is controllable, and can be targeted to produce a shared library that has a SystemC compatible interface.

9.4 Integration

In order for these two very different simulators to communicate it was necessary to reach some common ground. By basing the integration on SystemC [Automation, Design and Committee, Standards, 2006], we can utilize existing SystemC signal based interfaces to communicate between the ADL processor and the

Lyra PCI Express. Additionally, as these interfaces are standard, other simulators could be similarly bridged into the system. Figure 9·7 shows a block diagram overview of how the models communicate. Both models are driven by a shared clock and communicate using a serial protocol via a SystemC signal based interface. The interface consists of two sets of signals to allow for bidirectional communication and contain both control and data signals. Both the Lyra and ADL portions of the system simulator contain a few simple processes to convert the data to and from the serial form. The reason for this serial interface is that it alleviates mismatches in size between the ADL memory accesses and the PCI Express packets. As a result, it is more effective to transmit the data serially. Additionally, as the Lyra model may be synthesizable into hardware, it is beneficial to have a simple serial protocol that is area efficient. The protocol supports data sent in small increments, such as four bytes at a time, where the first 2 bytes define the length of the remaining message. In this implementation, the TLP packet is sent as the body of the message. There are other equally valid ways to implement the same functionality.

In the Lyra portion, this interface is based purely on shared register and signal communication. Using a tool that accepts Lyra models and creates SystemC equivalents, the model can be synthesized. Once synthesized to SystemC, these top level ports are presented as SystemC signal input or output ports. Two separate input and output processes buffer the data that has been received or is to be transmitted. As these processes are not rendezvous based, a third process is needed to communicate with the remainder of the system via rendezvous.

For the ADL portion of this interface, we use custom written function calls that replicate the same behavior as the Lyra case. We created a separate external memory module that could communicate with both the main memory and the ADL processor via a function call based interface and to the Lyra model via the SystemC based

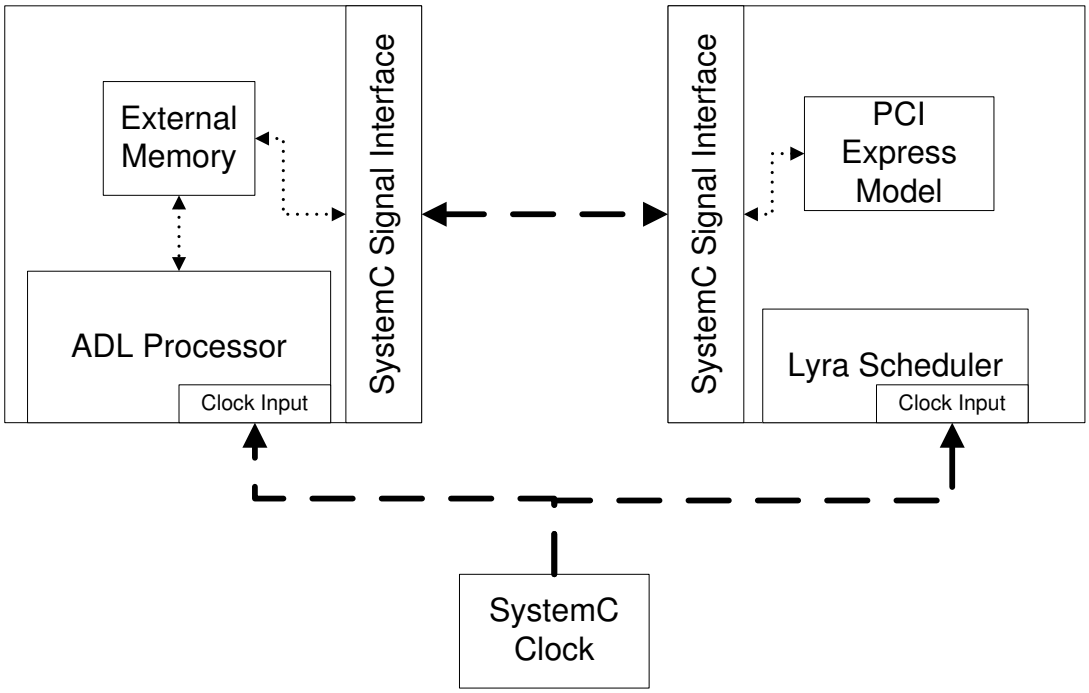

Figure 9·7: Integrated System Overview

serial interface. This external memory module takes the place of the normal ADL memory and intercepts all calls to main memory. In case of calls that attempt to access certain ranges of memory, they are transferred to the Lyra PCI Express model via the SystemC serial protocol. When calls are completed in Lyra, they are sent to the memory over the SystemC serial protocol, which returns the responses to the processor model.

As the clock is shared, the two models proceed in lock step. The shared clock requirement could be relaxed in the future by making the protocol fully asynchronous, in order to allow the simulators to proceed completely independently.

9.5 Results

The final PCI Express model in Lyra contained 26 processes and 20 rendezvous for the entire network. The ADL model consists of a 5 stage in order pipelined processor that accepts a PowerPC like instruction set. A small program was created that would write 8 bytes of data to an address mapped to a PCI Express endpoint, and read it back into the processor's memory. The resulting system simulates at 15 cycles/sec. As all the simulators share the same clock, the estimation of work done by the resulting simulator varies. For the ADL simulator, a SystemC clock tick corresponds to one cycle for the processor. For the Signal interface, a SystemC clock tick corresponds to the transfer of 4 bytes of data. In the Lyra model, a SystemC clock tick corresponds to the computation of a new schedule. This schedule could involve as much as multiple transactions or as little as reading from a signal into a register. Allowing these three domains to have different clocks should improve the performance of the resulting simulator.

Chapter 10

Empirical Results

In this chapter, we describe the empirical results of using Lyra for system modeling. We examine models that cover a diverse array of modeling styles and abstractions. We quantify the impact of different scheduling algorithms on the communication.

10.1 Empirical Result Overview

The results of developing a new methodology can be hard to quantify. As a result, we will be examining the results in a few separate pieces. Firstly, the primary problem with the adoption of rendezvous based approaches has been the question of search space that must be traversed at each run time iteration. Thus, a valid quantitative measure is to compare the improvement (i.e. reduction in search space) for the scheduler compared to a naive brute force scheduler and to an improved scheduler. In addition, since the scheduler is extremely model dependent, we can consider the practical results using these models.

10.2 Models

We performed tests on the following models, which represent a wide variety of modeling styles and abstraction levels. A short discussion of each of these models follows.

1. Elastic Buffer Pipeline of N stages (EP N)

2. PCI Express Simplified model (PEX)

3. Synchronous MIPS (SyncMIPS)

4. Asynchronous MIPS (AsyncMIPS)

5. Software JPEG encoder (JPEG)

10.2.1 Elastic Buffer Pipeline

The elastic pipeline concept was first introduced in [Cortadella et al., 2006]. This is a concept that was brought from the field of asynchronous design into the design of systems with multiple clocks. The notion of elasticity refers to the variable latency that is experienced by data going into the module, based on whether the module connected to the output is ready to accept a new piece of data. In a multi clock system, such a buffer, or pipeline of buffers can help a producer and consumer running at different clock rates communicate. As modern system-on-chips contain multiple cores running at different clock rates, an elastic buffer based pipeline can be used in many places to allow them to communicate. The problem with modeling these using traditional methodologies is that the variable latency of the blocks, and complex communication patterns possible, make it difficult to design a scalable implementation. To implement this in Lyra, a basic block was created that presents an input rendezvous and an output rendezvous. The module contains a single register of the same data type as the input and output. If the register is empty, and both input and output rendezvous can fire, then the data is passed straight from the input to the output, bypassing the register. If the output rendezvous cannot fire, but the input rendezvous is requested, the data is buffered. If the buffer is full, then the input is not allowed to fire either. If the output rendezvous alone occurs while there is data, it is sent on the rendezvous, otherwise, the output is not allowed. Finally, if the buffer is occupied, but both input

and output rendezvous can fire, then the behavior is like a pipeline stage, where the input data is stored into the buffer, and the previous contents of the buffer are sent on the output rendezvous. This basic block is then chained multiple times to create a buffer pipeline of a fixed length.

10.2.2 PCI Express Simplified

The PCI Express protocol is discussed in much more depth in Section 9.2.1. In this simplified implementation, the finite state machines are a lot simpler and implement a subset of the PCI Express space. In addition, the simplified model mainly tests the routing correctness of the model and as a result, the data payload of each TLP packet is simply set to 4 bytes. Finally, the testing of the PCI Express model occurs by integrating it with dummy endpoints and a specially designed root complex that generates a simple test pattern. Another very important fact is that this simplified PCI Express model only contains the bare essential portions of the root complex, and is self sufficient. Thus, there is no external interface mechanism to load data to and from the root complex. Finally, the model is preconfigured. In the PCI Express specification, each module is configured by setting data in its configuration address space, however, as the simplified model does not really implement the data handling, the modules are configured using template parameters.

10.2.3 Synchronous MIPS

This is an implementation of a 5 stage pipelined, 32 bit MIPS processor, much like in [Patterson and Hennessy, 2009]. As we are using a higher abstraction level design tool, we use a similar approach to modeling the system. We have 5 processes, one for each stage of the pipeline, and use rendezvous to represent all communications between the processes. For this model, we attempted to try to model at a lower

abstraction level. This was done by basing the model on the RTL implementation seen in [Patterson and Hennessy, 2009]. Each FSM contained the combinational logic for that stage, as well as the pipeline register corresponding to that stage. In general, the rendezvous outputs of the stage would be simply reading from the register, but rendezvous inputs to the stage would be making changes to the FSM. The usage of rendezvous, and their synchronous nature, allows for easy modeling of data forwarding and combinational behavior. At the same time, rendezvous used in conjunction with registers can give a good approximation of RTL. As a result, the resulting design is almost a transactional model of an RTL implementation. The reason this model is being called synchronous is that the model posses very tight coupling of rendezvous, i.e. there is an external rendezvous simulating a clock with is conjoined with most edges. As a result, the system is forced to synchronize very often and there is a lot of mutual exclusion that must be resolved by the scheduler.

10.2.4 Asynchronous MIPS

The asynchronous MIPS model is a variant of the synchronous MIPS model discussed in Section 10.2.3. Like SyncMIPS, the asynchronous model posses 5 pipeline stages, where each stage is modeled as a process, with rendezvous communication between them. The main difference between this and the synchronous model is that this model follows the asynchronous style of modeling. As a result, each stage has loose coupling of rendezvous. By using extra stages in the FSMs, many conjunctions are avoided. This has a noticeable effect in simplifying the ORG, and reducing the work that the scheduler needs to do. As might be clear, the performance of this approach is slightly worse than that of SyncMIPS as the data will have a higher latency due to the extra stages in the FSM.

10.2.5 JPEG Encoder

The JPEG encoder is a software dataflow style model. This model cannot be synthesized into SystemC due to its dependence on software functions to read and write to files. It implements the standard expressed in [Wallace, 1992]. The various stages of the data flow are implemented as separate FSMs, communicating via rendezvous. The data is read into a temporary buffer from a file by an input FSM. Next, the DCT is performed by another FSM. The next FSM in the chain performs the quantization, after which the cofficients are calculated. The next stages perform Huffman coding. Finally, the data is written to a file by a an FSM, which sends a signal back to the first machine, signalling it ready to accept a new file.

10.3 Scheduling approaches

While there are very few approaches that attempt to use the communication scheduling of their system to resolve concurrency and race conditions, the increasing complexity of communication makes it important. The problem of a metric for results of high level synthesis is an open one. Run time alone is not a good measure of a new methodology. Important factors include the "ability to search a large portion of the design space" [McFarland et al., 1990], the ability to synthesize models in addition to the flexibility of the methodology. Thus, a primary focus shall be the search space that must be examined at each run time iteration, i.e. the work the communication scheduler must do.

As a baseline for comparison, we consider a Brute Force Scheduler (BFS) algorithm that evaluates all combinations of rendezvous that can occur at each run time iteration, and, for each rendezvous, tests all TE combinations that can cause that rendezvous to occur. The steps of the brute force algorithm (BFS) are shown as pseudocode. It receives as input the set of all rendezvous R, and the current state S

of all processes and returns the best subset of rendezvous to fire. The *screen* function filters the rendezvous set R based on the current state information. It removes those rendezvous that do not appear on the edges from the current states.

The Brute Force Scheduler (BFS) shown accepts a set of rendezvous, and screens it based on the current state. It then iterates through all subsets of the rendezvous set. In each iteration, the subset is tested to see if it can indeed occur. In order to perform this test (by the test function above), all combinations of the "+" and "-" labels of all rendezvous in the selected subset must be checked to see whether the selected set can occur jointly. If the subset can occur, and the total weight of the subset is higher than the maximum weight seen so far, then the subset and the weight are stored. At the end, if a valid subset had been found, it occurs and the system transitions to the next state. In general, for each bi-party rendezvous, the algorithm checks m*n occurrence possibilities, if it has m labels of role "+" and n labels of role "-", and 1 non-occurrence possibility. Similarly, a barrier has bj occurrence possibilities where bj is the count of its labels of role j, and 1 non-occurrence possibility.

```
BFS (R, S)
Rs = screen(R, S)
maxWeight = −∞
maxComb = φ
for all Rt ⊆ Rs do
  if total weight of Rt ¿ maxWeight & test(Rt, S) then
    maxWeight = total weight of Rt
    maxComb = Rt
  end if
end for
return maxComb
```

As a simple improvement, we consider the use of state based pruning where TEs that are not incident to the current state are removed from consideration. As the removal of some TEs may cause the elimination of a role of a rendezvous, all TEs bearing other roles of the given rendezvous must be eliminated. As these new deletions may result in further removals, this procedure is run recursively until there are no more unready TEs left. A previous method of utilizing relationships between rendezvous to improve BFS was presented in [Wang et al., 2009].This approach, called Guided BFS (GBFS), is used as the second baseline for comparison. The Guided Brute Force Scheduling (GBFS) algorithm accepts a rendezvous decision tree, which contains data about how to decide between rendezvous using weight. The rendezvous decision tree contains 3 types of nodes - decision nodes, which decide between the two subtrees, join nodes, that imply concurrence of subtrees and leaf nodes. The leaf nodes contain sets of related rendezvous, i.e. the product space of the TEs of all related rendezvous. These rendezvous relations are extracted through simple state analysis of the original description. This tree is sent to the GBFS algorithm which then prunes it to remove branches and vertices it knows cannot occur under the current state. The algorithm then examines the tree in a top down manner, starting at the root node. Whenever a join node is encountered, the algorithm can evaluate each child separately and combine the best results from the children. When a decision node is encountered, the scheduler evaluates both branches and selects the better result between them. Only when a leaf node is encountered, the BFS algorithm is used to evaluate the set of vertices at the node.

```
GBFS(T, S)
prune(T, S)
if root(T) is a decision node then
   comb1 = GBFS(taken child, S) ∪ {root(T)}
   comb2 = GBFS(untaken child, S)
   return best of comb1 and comb2
else if root(T) is a join node then
   result = φ
   for all child Ti do
      result = result GBFS(Ti, S)
      return result
   end for
else
   return BFS(R(T))
end if
```

10.4 Result Summary

In the current implementation of the scheduler, used to generate the results, we make the assumption that the system behaves in a synchronous manner. Thus, every process is assumed to behave synchronously and is at a fixed state for each scheduling iteration. For performance metrics, we can compare the average search space per run time iteration for each model. For the BFS and GBFS approaches, we define the search space as the average number of schedules per step. For the ORG based approach, we define the search space as the average number of minimal schedules that are activated at each step. Furthermore, the current version of the algorithm has been implemented in the software simulator and the results are summarized in

Model	BFS	GBFS	ORG
EP 2	4	4	2.5
EP 4	28	28	3.5
EP 8	2188	2188	5.5
EP 16	14348908	14348908	9.5
PEX	416286	703	1.6
SyncMIPS	346420800	208285	1
AsyncMIPS	155	16	2.56
JPEG	6563	7	2.15

Table 10.1: Scheduler Search Spaces

Table 10.1.

As we can note, the search space for ORG based algorithms is the same. However, the choice of policy affects the number of schedules selected. An interesting fact to note is that the search space for expands with factorial complexity in the presence of a large degree of conjunction, that neither of the previous approaches (i.e. BFS and GBFS) could handle well. The best example of this is the elastic buffer pipeline. As each stage contains a combinational path, a pipelining path and a read only and write only path, there is a large amount of conjunctive composition in each rendezvous. As the pipeline must allow for bypass of all empty buffer stages, potentially of n stages in an n stage pipeline, there exist some long combinational paths in the resulting system. The search space for the BFS based approach is factorial as it must compute the power set of all possible sets of transition edges, assuming forwarding to the nth stage, the n-1th stage etc.

In this scenario, the GBFS approach, which relies on rendezvous relations fails as it cannot determine the exact nature of the relationships. The benefit of the ORG based

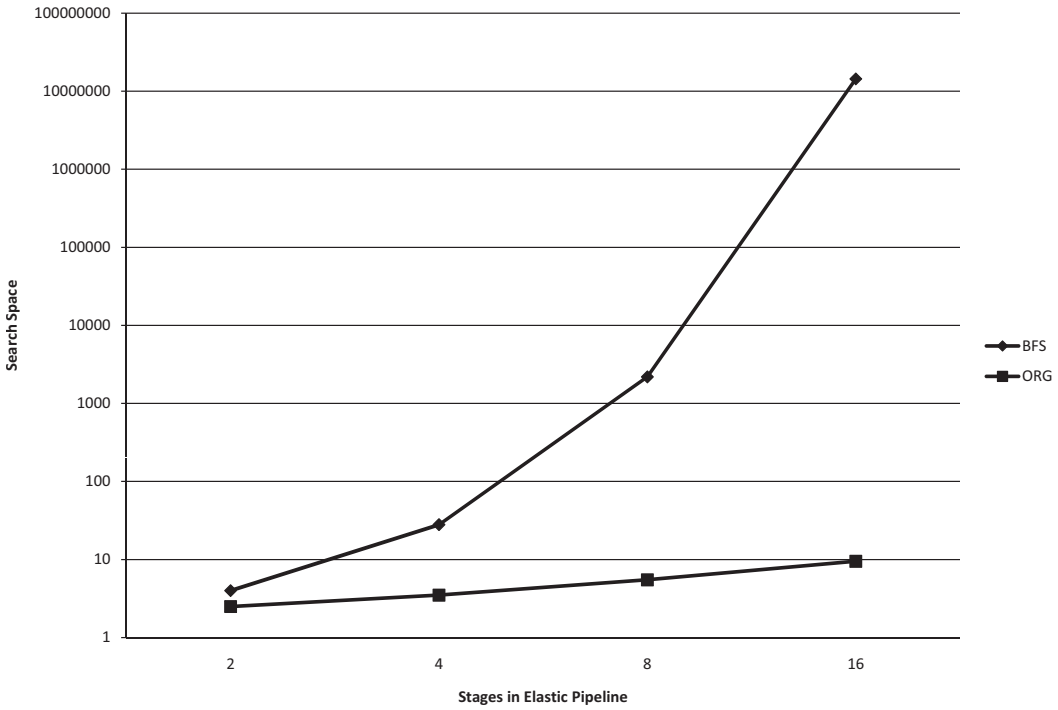

Figure 10·1: Complexity comparison

Model	BFS	GBFS	ORG
EP 2	9	16	8
EP 4	32	60	17
EP 8	387	736	47
EP 16	29325	55784	155
PEX	697	1345	70
SyncMIPS	120	239	120
AsyncMIPS	45	84	28
JPEG	7686	13440	93

Table 10.2: Scheduler Search Spaces

scheduling mechanism is clearly shown in Figure 10·1. The approach using the ORG based scheduler grows almost linearly, as opposed to the BFS based approach that is super linear. As the other models do not posses this high a degree of conjunction, the GBFS approach presents some benefits, but the ORG based approach is superior compared to it.

The storage space needed for each policy is summarized in Table 10.2.

The benefits of using the ORG based scheduler approach are clear. The general comparison was performed using the GWO approach, which solves the same problem and presents the same solution as the BFS and GBFS approaches. By the determination of dependencies between transition edges in a static manner, we can reduce the space that must be examined at each run time iteration.

Chapter 11

Conclusions and Future Work

11.1 Conclusion

The need for a high level methodology for the design of modern computer systems has been shown. Presently, there has been no agreement on the approach necessary to replace RTL. While many different tools and flows are currently being developed, the most popular among them - Transaction Level Modeling based on SystemC has some fundamental shortcomings. The extension of a software programming language to hardware bring with it well studied problems of modeling concurrent systems using a sequential approach. The lack of a formalization for this approach complicates the creation of hardware from such models. The lack of a good abstraction for communication does not allow for the modeling of complex communication patterns.

This work demonstrates a novel methodology for the high level modeling of systems using a rendezvous based communication abstraction. This methodology has advantages compared to previous rendezvous based approaches and to the TLM/SystemC approach. Compared to other rendezvous based approaches, this methodology can allow for the modeling of complex communication patterns as it allows the full and free composition of rendezvous using the conjunction and disjunction primitives. The presence of nondeterminism in the resulting model allows for the expression of complex models with partial descriptions, reducing the work done by the designer and raising the level of abstraction. The presence of an underlying

formal basis makes this approach much more attractive compared to TLM/SystemC.

The novel nondeterminism resolution mechanism allows for the creation of synthesis friendly systems, while retaining the speed benefits of using a higher level approach. The scheduler framework is shown to be extensible, allowing for future extensions to different tradeoffs between size and complexity of the nondeterminism resolution heuristic. The general synthesis flow for Lyra is shown to be functional, while retaining independence between the scheduler and the system.

The graph based tools developed for the approach demonstrate their usefulness in analyzing other approaches that are based on atomic actions and their composition. Further, they demonstrate a formal method of construction of a scheduler, from an input design. This construction creates a NFA, which has been shown to be exponentially smaller in complexity than a deterministic scheduler. The use of the scheduling policy as a heuristic to create a DFA from the NFA is also shown and the two policies are examined in terms of the construction complexity.

While the basic work has been done to create a viable high level modeling and synthesis flow, there are many aspects that remain to be addressed. The extension of the formal model for verification and model checking. Design verification is an increasingly important area. The presence of a formal basis for Lyra makes it much more capable of performing in-depth formal analyses of models compared to ad hoc approaches like SystemC. The creation of tools for symbolic model checking and formal verification of Lyra models is a logical next step in this approach. Another area that can be addressed is the creation of a performance optimization suite for Lyra. The use of a high level design give rise to opportunities for powerful design optimizations. As the complexity of Lyra models can be analyzed using the graph based approaches, a performance analysis tool could help designers by giving design hints. As a simple example, the presence of high concurrency has been shown to increase

the size of the scheduler and the complexity of the design. A sample optimization would be then be the detection in the TRG of the solid edges which, when removed, cause the maximum number of solidly disconnected components. These edges could then be presented to the designer as conjunctions that could be sources for removal.

Lyra presents a new methodology for the design of modern systems. By expanding the power of rendezvous, it allows the designer the freedom to design complex systems, in a simple fashion. The familiar finite state machine based description makes it easy for hardware designers to utilize the language, while the rendezvous abstraction allows for simply expressing complex communication and synchronization patterns. The presence of synchronous dataflow and combinational signals allows for pure dataflow modeling. The novel communication scheduler ensures the resulting design is synthesizable in a transparent manner. The choice of scheduling policy allows the designer to retain control and provide hints. The synthesis friendly design ensures that the models remain synthesizable. The approach has been demonstrated using a series of practical models that exhibit a wide variety of modeling styles and levels of abstraction.

References

[Accellera Ltd, 2004] Accellera Ltd (2004). *SystemVerilog 3.1 Language Reference Manual*.

[ANSI, 1989] ANSI (1989). Standard X3. 159-1989, Programming Language C, ANSI. *Inc., NY*.

[Armoni and Ben-Ari, 2009] Armoni, M. and Ben-Ari, M. (2009). The concept of nondeterminism: Its development and implications for education. *Science & Education*, 18(8):1005–1030.

[Automation, Design and Committee, Standards, 2006] Automation, Design and Committee, Standards (2006). IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2002(March):16662005.

[Barbacci, 1975] Barbacci, M. (1975). A comparison of register transfer languages for describing computers and digital systems. *IEEE Transactions on Computers*, C-24(2):137 – 150.

[Barret, 1992] Barret, G. (1992). Occam3 Reference Manual. *Inmos Lmtd*.

[Berry and Gonthier, 1992] Berry, G. and Gonthier, G. (1992). The E synchronous programming language: design, semantics, implementation* 1. *Science of Computer Programming*, 19(2):87–152.

[Bolognesi and Brinksma, 1987] Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59.

[Budruk et al., 2003] Budruk, R., Anderson, D., and Shanley, T. (2003). *PCI express system architecture*. Addison Wesley Publishing Company.

[Cadence Inc, 2008] Cadence Inc (2008). C-to-silicon compiler datasheet.

[Cai and Gajski, 2003] Cai, L. and Gajski, D. (2003). Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, page 24. ACM.

[Camposano and Wolf, 1991] Camposano, R. and Wolf, W. (1991). *High-level VLSI synthesis*. Kluwer Academic Publishers Norwell, MA, USA.

[Carrol and Long, 1989] Carrol, J. and Long, D. (1989). *Theory of finite automata*. Prentice-Hall Englewood.

[Celoxica Ltd, 2003] Celoxica Ltd (2003). Handel-C language reference manual. *Document Number: RM-1003-4.2*.

[Chandra et al., 1981] Chandra, A., Kozen, D., and Stockmeyer, L. (1981). Alternation. *Journal of the ACM (JACM)*, 28(1):114–133.

[Chandra and Stockmeyer, 1976] Chandra, A. and Stockmeyer, L. (1976). Alteration. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science, Houston, Texas*, pages 98–108.

[Cortadella et al., 2006] Cortadella, J., Kishinevsky, M., and Grundmann, B. (2006). Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd annual Design Automation Conference*, page 662. ACM.

[De Micheli, 1999] De Micheli, G. (1999). Hardware synthesis from c/c++ models. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 80, New York, NY, USA. ACM.

[de Wit and Peeters, 2006] de Wit, M. and Peeters, A. (2006). Haste language reference manual. Technical report, Handshake Solutions.

[Densmore et al., 2006] Densmore, D., Passerone, R., and Sangiovanni-Vincentelli, A. (2006). A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers*, 23(5):359–374.

[Dijkstra, 1975] Dijkstra, E. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

[Donlin, 2004] Donlin, A. (2004). Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA. ACM.

[Drusinsky and Harel, 1994] Drusinsky, D. and Harel, D. (1994). On the power of bounded concurrency i: finite automata. *Journal of the ACM*, 41(3):517–539.

[Edwards, 2005a] Edwards, S. (2005a). The challenges of hardware synthesis from C-like languages. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 66–67.

[Edwards et al., 2001] Edwards, S., Lavagno, L., Lee, E., and Sangiovanni-Vincentelli, A. (2001). *Readings in hardware/software co-design*, chapter Design of embedded systems: Formal models, validation, and synthesis. Morgan Kaufmann.

[Edwards and Tardieu, 2005] Edwards, S. and Tardieu, O. (2005). SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM International Conference on Embedded Software*, page 272. ACM.

[Edwards, 2005b] Edwards, S. A. (2005b). The challenges of hardware synthesis from c-like languages. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 66–67, Washington, DC, USA. IEEE Computer Society.

[Ernst et al., 1996] Ernst, R., Henkel, J., Benner, T., Ye, W., Holtmann, U., Herrmann, D., and Trawny, M. (1996). The COSYMA environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159–166.

[Evangelist et al., 1989] Evangelist, M., Francez, N., and Katz, S. (1989). Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15:1417–1426.

[Fujita and Nakamura, 2001] Fujita, M. and Nakamura, H. (2001). The standard SpecC language. In *Proceedings of the 14th international symposium on Systems synthesis*, page 86. ACM.

[Gajski and Ramachandran, 1994] Gajski, D. and Ramachandran, L. (1994). Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54.

[Gajski et al., 2000a] Gajski, D., Wu, A., Chaiyakul, V., Mori, S., Nukiyama, T., and Bricaud, P. (2000a). Essential issues for IP reuse. In *Proceedings of the Asia South Pacific Design Automation Conference*, pages 37–52.

[Gajski et al., 2000b] Gajski, D., Zhu, J., Domer, R., Gerstlauer, A., and Zhao, S. (2000b). *SpecC: Specification Language and Methodology*. Springer Netherlands.

[Gajski et al., 1992] Gajski, D. D., Dutt, N. D., Wu, A. C.-H., and Lin, S. Y.-L. (1992). *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, Norwell, MA, USA.

[Galloway et al., 1995] Galloway, D. et al. (1995). The transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144.

[Gerstlauer et al., 2005] Gerstlauer, A., Shin, D., Domer, R., and Gajski, D. (2005). System-level communication modeling for network-on-chip synthesis. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 45 – 48.

[Gupta and Brewer, 2008] Gupta, R. and Brewer, F. (2008). High-Level Synthesis: A Retrospective. *High-Level Synthesis*, pages 13–28.

[Hachtel and Somenzi, 1996] Hachtel, G. and Somenzi, F. (1996). *Logic synthesis and verification algorithms*. Kluwer Academic Pub.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320.

[Hemani, 2004] Hemani, A. (2004). Charting the eda roadmap. *IEEE Circuits and Devices Magazine*, 20(6):5 – 10.

[Hemani et al., 2000] Hemani, A., Jantsch, A., Kumar, S., Postula, A., Oberg, J., Millberg, M., and Lindqvist, D. (2000). Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*, pages 166–173.

[Hoare, 1978] Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):677.

[Hopcroft et al., 1979] Hopcroft, J., Motwani, R., and Ullman, J. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley Reading, MA.

[IEEE, 2000] IEEE (2000). *IEEE Std 1076-2000 : Standard VHDL Language Reference Manual*. IEEE.

[IEEE, 2001] IEEE (2001). IEEE Std 1364-2001 : Draft Standard for Verilog Hardware Description Language.

[Joung and Smolka, 1996] Joung, Y. and Smolka, S. (1996). A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM (JACM)*, 43(1):75–115.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475.

[Kambe et al., 2001] Kambe, T., Yamada, A., Nishida, K., Okada, K., Ohnishi, M., Kay, A., Boca, P., Zammit, V., and Nomura, T. (2001). A C-based synthesis system, Bach, and its application (invited talk). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 151–155. ACM.

[Kernighan and Ritchie, 1988] Kernighan, B. and Ritchie, D. (1988). *The C programming language*. Prentice-Hall.

[Kozen, 1976] Kozen, D. (1976). On parallelism in Turing machines. In — *17th Annual Symposium on Foundations of Computer Science*, pages 89–97. IEEE.

[Ku and De Micheli, 1990] Ku, D. and De Micheli, G. (1990). HardwareC-a language for hardware design version 2.0. Technical report, Stanford University.

[Kumar et al., 2002] Kumar, S., Jantsch, A., Soininen, J., Forsell, M., Millberg, M., Oberg, J., Tiensyrja, K., and Hemani, A. (2002). A network on chip architecture and design methodology. In *IEEE Symposium on Very Large Scale Integration (VLSI)*, pages 117–124.

[Kurd et al., 2010] Kurd, N., Bhamidipati, S., Mozak, C., Miller, J., Wilson, T., Nemani, M., and Chowdhury, M. (2010). Westmere: A family of 32nm ia processors. In *2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 96 –97.

[Lee and Yannakakis, 1996] Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines. *Proceedings of IEEE*, 84:1090–1123.

[Liao et al., 1997] Liao, S., Tjiang, S., and Gupta, R. (1997). An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 70–75, New York, NY, USA. ACM.

[MacMillen et al., 2000] MacMillen, D., Butts, M., Camposano, R., Hill, D., and Williams, T. (2000). An industrial view of electronic design automation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1428–1448.

[Maillet-Contoz and Ghenassia, 2005] Maillet-Contoz, L. and Ghenassia, F. (2005). Transaction level modeling. In Ghenassia, F., editor, *Transaction Level Modeling with SystemC*, pages 23–55. Springer US.

[Man, 2005a] Man, K. (2005a). Formal communication semantics of SystemC FL. In *Proceedings of 8th Euromicro Conference on Digital System Design*, pages 338–345.

[Man, 2005b] Man, K. (2005b). SystemC FL: a formalism for hardware/software codesign. In *Proceedings of the 2005 European Conference on Circuit Theory and Design*. IEEE.

[Martin and Smith, 2009] Martin, G. and Smith, G. (2009). High-level synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26:18–25.

[McCloud, 2004] McCloud, S. (2004). Catapult c synthesis-based design flow: Speeding implementation and increasing flexibility. Technical report, Mentor Graphics.

[McFarland et al., 1990] McFarland, M., Parker, A., and Camposano, R. (1990). The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301 –318.

[Mead and Conway, 1980] Mead, C. and Conway, L. (1980). *Introduction to VLSI systems*. Addison-Wesley Reading, MA.

[Nikhil, 2008] Nikhil, R. (2008). Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. *High-Level Synthesis*, pages 129–146.

[Patterson and Hennessy, 2009] Patterson, D. and Hennessy, J. (2009). *Computer organization and design: the hardware/software interface*. Morgan Kaufmann Pub.

[Pestana et al., 2004] Pestana, S., Rijpkema, E., Rdulescu, A., Goossens, K., and Gangwal, O. (2004). Cost-performance trade-offs in networks on chip: A simulation-based approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, volume 2. IEEE Computer Society.

[Rabin and Scott, 1959] Rabin, M. and Scott, D. (1959). Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125.

[Sangiovanni-Vincentelli, 2003] Sangiovanni-Vincentelli, A. (2003). The tides of EDA. *IEEE Design & Test of Computers*, 20(6):59–75.

[Semeria and De Micheli, 1998] Semeria, L. and De Micheli, G. (1998). Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 340–346, New York, NY, USA. ACM.

[Soderman and Panchul, 1998] Soderman, D. and Panchul, Y. (1998). Implementing C algorithms in reconfigurable hardware using C2Verilog. In *IEEE Symposium on FPGAs for Custom Computing Machines, 1998. Proceedings*, pages 339–342.

[Stroud et al., 1988] Stroud, C., Munoz, R., and Pierce, D. (1988). Behavioral model synthesis with cones. *IEEE Design and Test of Computers*, pages 22–30.

[Taft and Duff, 1997] Taft, T. and Duff, R. (1997). *ADA 95 Reference Manual Language and Standard Libraries*. Springer-Verlag.

[Venkataraman et al., 2009a] Venkataraman, V., Di Wang, W., Bose, M., and Bhadra, J. (2009a). Simulation of a Heterogeneous System at Multiple Levels of Abstraction Using Rendezvous Based Modeling. In *10th International Workshop on Microprocessor Test and Verification*, pages 3–8. IEEE.

[Venkataraman et al., 2009b] Venkataraman, V., Wang, D., Mahram, A., Qin, W., Bose, M., and Bhadra, J. (2009b). Synthesis Oriented Scheduling of Multiparty Rendezvous in Transaction Level Models. In *Proceedings of the 2009 IEEE Computer Society Annual Symposium on VLSI-Volume 00*, pages 241–246. IEEE Computer Society.

[Wakabayashi, 1999] Wakabayashi, K. (1999). C-based synthesis experiences with a behavior synthesizer. In *Proceedings of Design, Automation and Test in Europe*, page 390. Published by the IEEE Computer Society.

[Wakabayashi, 2004] Wakabayashi, K. (2004). C-based behavioral synthesis and verification analysis on industrial design examples. In *ASP-DAC '04: Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 344–348, Piscataway, NJ, USA. IEEE Press.

[Wallace, 1992] Wallace, G. (1992). The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):18 – 34.

[Wang et al., 2009] Wang, D., Venkataraman, V., Wang, Z., Qin, W., Wang, H., Bose, M., and Bhadra, J. (2009). Accelerating multi-party scheduling for transaction-level modeling. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 339–344. ACM.

CURRICULUM VITAE

VYAS VENKATARAMAN

2701 SAN TOMAS EXPY • SANTA CLARA • CA 95050

E-MAIL: VYAS_V@IEEE.ORG

EDUCATION

College of Engineering, Boston University, Boston, MA

PhD, Computer Engineering

College of Engineering, Boston University, Boston, MA

Master of Sciences, Computer Engineering, September 2008

College of Engineering, Boston University, Boston, MA

Bachelor of Sciences, Computer Systems Engineering, May 2006

Summa Cum Laude

AWARDS RECEIVED

- Best Paper award at DATE'08
- Received Dean's Research Fellowship in 2006
- Finalist at IEEE Computer Society International Design Competition (CSIDC) 2006, with senior project – SmarTrash – "A wireless mesh network of trashcans to protect urban environments"
- GE Imagination Award 2006 at Boston University for imagination and creativity
- Represented India at IEEE International Science and Engineering Fair with high school project "Red Shift : a 3D design tool" as a result of winning the "All - India Science Fair, Team Event", 1999

WORK EXPERIENCE

Developer tools, Nvidia Inc 2010-present

CUDA Software engineer

- Primary engineer on CUDA-MEMCHECK, a multi-platform run time functional correctness checking tool for parallel programs
- Worked on CUDA-GDB, a port of the GNU debugger with support for debugging CUDA programs

Electrical and Computer Engineering Dept at Boston University 2008-2010

Graduate Researcher

- Worked on creating a high level modeling approach using rendezvous
- Created a custom language, synthesis and simulation tools implementing this approach

Tools and Methodology Group, Freescale Semiconductor Inc Summer 2008

Student Technical Intern

- Worked on a high level model of a system on chip
- Created a bridge between two simulators and a model of PCI Express Network

Reliable Computing Laboratory at Boston University 2006-2008

Graduate Researcher

- Worked on creating EDA flow for circuit synthesis from high level description
- Worked on capacitance invariant dual rail power balanced library resistant to side channel attacks.

Electrical and Computer Engineering Dept at Boston University 2007-2008

Graduate Teaching Fellow

- Created homework assignments, lead lab sections and guided students through a project for the undergraduate computer architecture course at BU
- Functioned as part of the teaching team, supervising graders and Undergraduate Teaching Fellows.

Networked Information Systems Lab at Boston University Summer 2005

Undergraduate Researcher

- Developed a fast and bandwidth efficient reconciliation protocol implementation competitive with the industry standard rsync protocol

PROJECTS

SmarTrash 2005-2006

Senior Design Team Project

- Created a mesh network of crossbow MICA 2 motes with fully custom networking algorithm, custom PCB and power circuitry
- Was one of the 10 finalists in the CSIDC 2006 competition, one of three representing USA

pxaDoom Fall 2007

Embedded Systems

- Created a custom linux distribution for PXA 270 development board and a custom Linux PS/2 driver to play Doom on the PXA 270 as part of final project for Embedded Systems class.

Neighborhood Coincidence Processing Unit for LIDAR system Fall 2005

FPGA and Advanced Digital Design

- Created a Verilog based hardware implementation of Neighborhood coincidence processing algorithm for de-noising output of a LIDAR system.

LEADERSHIP POSITIONS

- Officer of Eta Kappa Nu
- Officer of Boston University IEEE chapter
- Resident Assistant at Boston University
- Deans Host at Boston University College of Engineering

PROFESSIONAL MEMBERSHIPS

- Member of the IEEE (Silicon Valley Chapter)
- Eta Kappa Nu (Electrical and Computer Engineer's Honor Society)

PUBLICATIONS

- V. Venkataraman, D. Wang, W. Qin, M. Bose, J. Bhadra, "Simulation of a heterogeneous system at multiple levels of abstraction using Rendezvous based modeling", *Microprocessor Testing and Verification. MTV 2009*
- V. Venkataraman, D. Wang, W. Qin, M. Bose, J. Bhadra, "Synthesis-oriented scheduling of multiparty rendezvous in Transaction Level Models", *International Symposium on VLSI, 2009. ISVLSI 2009*
- D. Wang, V. Venkataraman, Z. Wang, W. Qin, M. Bose, J. Bhadra, "Accelerating Multi-party Scheduling for Transaction-level Modeling", *Great Lakes Symposium on VLSI 2009. GLSVLSI 2009*
- K. Kulikowski, V. Venkataraman, Z. Wang, A. Taubin, M.G. Karpovsky, "Asynchronous Balanced Gates Tolerant to Interconnect Variability", *International Symposium on Circuits and Systems, 2008. ISCAS '08*
- K. Kulikowski, V. Venkataraman, Z. Wang; A. Taubin, "Power Balanced Gates Insensitive to Routing Capacitance Mismatch," *Design, Automation and Test in Europe, 2008. DATE '08(Awarded Best Paper)*

AREAS OF INTEREST

- High Level Modeling and Synthesis
- Parallel computing
- Embedded Systems
- Electronic Design Automation
- Secure Hardware