

UNIVERSITY OF CALIFORNIA,  
IRVINE

**Functional Verification of System Level Model Refinements**

DISSERTATION

submitted in partial satisfaction of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Samar Abdi

Dissertation Committee:  
Professor Daniel D. Gajski, Chair  
Professor Ian G. Harris  
Professor Rainer Doemer  
Professor Tony Givargis

2005



The dissertation of Samar Abdi  
is approved and is acceptable in quality  
and form for publication on microfilm:

---

---

---

---

Committee Chair

University of California, Irvine  
2005

DEDICATION

*To my parents*

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>Acknowledgments</b>	<b>ix</b>
<b>Curriculum Vitae</b>	<b>x</b>
<b>Abstract of the Dissertation</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Traditional System Verification . . . . .	2
1.1.1 Pin and Cycle Accurate Simulation . . . . .	3
1.1.2 Transaction Level Simulation . . . . .	5
1.2 Proposed System Verification . . . . .	6
1.2.1 System Design Methodology . . . . .	6
1.2.2 Formal Modeling . . . . .	7
1.2.3 Verification by Correct Refinement . . . . .	7
1.3 Thesis organization . . . . .	8
<b>2 System Verification Techniques</b>	<b>9</b>
2.1 Simulation based Methods . . . . .	10
2.1.1 Stimulus Optimization . . . . .	11
2.1.2 Monitor Optimization . . . . .	12
2.1.3 Speed up techniques . . . . .	12
2.1.4 Modeling Techniques . . . . .	13
2.2 Formal Verification Methods . . . . .	14
2.2.1 Logic Equivalence Checking . . . . .	14
2.2.2 FSM Equivalence Checking . . . . .	15
2.2.3 Model Checking . . . . .	17
2.2.4 Theorem Proving . . . . .	19
2.2.5 Drawbacks of Formal Verification . . . . .	22
2.2.6 Improvements to Formal Verification Methods . . . . .	22
2.2.7 Semi-Formal Methods: Symbolic Simulation . . . . .	23
2.3 Evaluation Metrics for Verification Techniques . . . . .	23
2.4 Future of System Verification . . . . .	24

2.5	Chapter Summary . . . . .	25
<b>3</b>	<b>Model Algebra</b>	<b>26</b>
3.1	Definition . . . . .	26
3.1.1	Objects . . . . .	27
3.1.2	Composition Rules . . . . .	28
3.2	Model Construction with MA . . . . .	31
3.2.1	Hierarchy . . . . .	31
3.2.2	Parallel and Conditional Execution . . . . .	32
3.2.3	Variable Access via Ports . . . . .	33
3.2.4	Channel Access via Ports . . . . .	34
3.2.5	Using Identity Behaviors . . . . .	35
3.2.6	Hierarchical Modeling in MA . . . . .	37
3.3	Execution Semantics . . . . .	39
3.3.1	Behavior Control Graph . . . . .	39
3.3.2	Channel Semantics . . . . .	41
3.3.3	Control paths and dominators . . . . .	43
3.4	Notion of Functional Equivalence . . . . .	45
3.4.1	Value Traces . . . . .	45
3.4.2	Trace based equivalence . . . . .	47
3.5	Transformation laws of MA . . . . .	48
3.5.1	Flattening of Hierarchical Behaviors . . . . .	48
3.5.2	Control flow resolution of links . . . . .	51
3.5.3	Variable merging . . . . .	54
3.5.4	Identity elimination . . . . .	55
3.5.5	Redundant control dependency elimination . . . . .	60
3.5.6	Control relaxation . . . . .	61
3.5.7	Streamlining . . . . .	63
3.6	Chapter Summary . . . . .	65
<b>4</b>	<b>Functionality Preserving Refinements</b>	<b>66</b>
4.1	Refinement Based Design Methodology . . . . .	66
4.2	Behavior Partitioning . . . . .	69
4.2.1	Design Decisions . . . . .	69
4.2.2	Refinement Algorithm for Lock-Step output . . . . .	70
4.2.3	Correctness Proof for Lock-Step Refinement . . . . .	72
4.2.4	Refinement Algorithm for RPC style output . . . . .	77
4.2.5	Correctness Proof for RPC Style Refinement . . . . .	79
4.3	Serializing . . . . .	83
4.3.1	Design Decisions . . . . .	84
4.3.2	Refinement Algorithm . . . . .	85
4.3.3	Proof of Correctness . . . . .	85
4.4	Communication Scheduling . . . . .	88
4.4.1	Design Decisions . . . . .	89
4.4.2	Refinement Algorithm . . . . .	89

4.4.3	Proof of Correctness . . . . .	90
4.5	Transaction Routing . . . . .	94
4.5.1	Design Decisions . . . . .	94
4.5.2	Refinement Algorithm . . . . .	95
4.5.3	Proof of Correctness . . . . .	96
4.6	Chapter Summary . . . . .	99
<b>5</b>	<b>Automatic Refinement Verification</b>	<b>101</b>
5.1	Functional Abstraction . . . . .	102
5.1.1	SpecC language and SIR . . . . .	103
5.1.2	Executable Performance Models in SpecC . . . . .	104
5.1.3	Deriving Model Algebraic Expression . . . . .	105
5.2	Model Normalization . . . . .	113
5.2.1	Normalization Algorithm . . . . .	113
5.3	Isomorphism Testing of Normalized Models . . . . .	115
5.4	Experimental Results . . . . .	117
5.5	Chapter Summary . . . . .	119
<b>6</b>	<b>Conclusion</b>	<b>121</b>
6.1	Benefits . . . . .	122
6.2	Contribution . . . . .	122
6.3	Future Directions . . . . .	123
	<b>Bibliography</b>	<b>125</b>

# List of Figures

1.1	2003 survey results for reasons behind chip respins . . . . .	2
1.2	Pin-Accurate System Model . . . . .	3
1.3	Cycle accurate simulation environment . . . . .	4
1.4	A transaction level model of the system . . . . .	5
1.5	An illustration of proposed verification methodology . . . . .	6
2.1	A typical simulation environment . . . . .	10
2.2	Simulation optimization using coverage feedback . . . . .	11
2.3	Graphical visualization of model execution . . . . .	12
2.4	System level models with different computation and communication details . . . . .	13
2.5	Logic equivalence checking . . . . .	14
2.6	FSM equivalence checking . . . . .	17
2.7	Model checking of a transition system . . . . .	17
2.8	Example of a computation tree . . . . .	18
2.9	A CMOS inverter implementing $y = \neg x$ . . . . .	20
3.1	Control flow within hierarchical behaviors . . . . .	32
3.2	(a)Parallel and (b)FSM style compositions of behaviors . . . . .	33
3.3	Using ports for non-blocking data flow in hierarchical behaviors . . . . .	34
3.4	Sharing channel for transactions with different addresses . . . . .	35
3.5	Various manifestations of the identity behavior . . . . .	36
3.6	A hierarchical behavior with local objects and relations . . . . .	37
3.7	Hierarchical behavior $b_{par}$ showing (a) local scope, and (b) all scopes down to leaf behaviors . . . . .	38
3.8	The firing semantics of BCG nodes . . . . .	40
3.9	Timing diagram of a transaction on a channel . . . . .	41
3.10	Multiple competing transactions on a single channel . . . . .	42
3.11	Illustration of control paths . . . . .	44
3.12	Example of a model ( $M$ ) with unique value trace . . . . .	46
3.13	Example of a model ( $M'$ ) with several possible value traces . . . . .	47
3.14	Illustration of flattening law . . . . .	48
3.15	Resolution of channels into control dependencies . . . . .	51
3.16	Variable merging . . . . .	53

3.17	Identity elimination rule . . . . .	56
3.18	Redundant control dependency elimination . . . . .	59
3.19	Control relaxation rule . . . . .	61
3.20	Streamlining rule . . . . .	63
4.1	Refinement based design methodology . . . . .	67
4.2	Lock-step style output of behavior partitioning refinement . . . . .	71
4.3	Proof steps for refinement to lock-step style model . . . . .	75
4.4	RPC style output of behavior partitioning refinement . . . . .	77
4.5	Proof steps for refinement to RPC style model . . . . .	81
4.6	Serialization of parallel behaviors . . . . .	85
4.7	Inductive proof steps for serialization of parallel behaviors . . . . .	87
4.8	Different communication schedules for transaction over channel $c_{bus}$ . . . . .	88
4.9	Proof steps for verification of communication scheduling . . . . .	92
4.10	Routing transactions via transducer . . . . .	95
4.11	Proof steps for transducer insertion during transaction routing refinement . . . . .	98
5.1	Refinement Verification methodology . . . . .	102
5.2	Steps in refinement verification . . . . .	103
5.3	The platform model and its corresponding functional abstraction . . . . .	105
5.4	Data structure for storing model algebraic expression . . . . .	106
5.5	Data structure for storing model algebraic expression ( <i>contd.</i> ) . . . . .	107
5.6	Matching graph nodes during isomorphism testing . . . . .	116
5.7	Performance of verification tool for various refinements . . . . .	118
5.8	Refinement path vs. proof path . . . . .	119
6.1	Classes of models in Model Algebra . . . . .	124

# Acknowledgments

I would like to take this opportunity to thank all the people who have contributed to the fruition of this dissertation. First of all, I would like to thank my advisor, Professor Daniel Gajski for excellent guidance during my PhD studies. He has taught me how to think about problems, how to approach the solution and when to commit to a solution. I am grateful for all the stimulating discussions in the coffee shop and during Saturday lunches. I will forever be indebted for the time and the effort he has spent in my education.

I would also like to thank Professor Rainer Doemer, who was very patient and helpful everytime I went to him for assistance with my software. Dr. Andreas Gerstlauer, who has an unfathomable knowledge of computer science and beyond, was always available for technical help. I had many lively discussions with Dr. Junyu Peng and Dr. Dongwan Shin and collaborated with them on several papers. I would also like to thank many friends in CECS, including Partha Biswas, Mehrdad Reshadi and Bitra Gorji-Ara, for their great company. Special mention must be made of the SICS staff, particularly Melanie Sanders and Milena Wypchlak, for all their help.

This thesis would not have been possible without the unflinching support of my wonderful wife, Jelena. She celebrated my success and lifted my spirits me whenever I faced rejection. She is by far the single most important reason why I maintained my emotional balance through the ups and downs of graduate life.

Finally, I would like to thank my parents, to whom this thesis is dedicated. My dear departed mother would have been very proud today. My father has made many personal sacrifices to provide me the best possible education and a healthy atmosphere at home. It is impossible to put down in words their contribution to my personal growth. I would also like to thank my brother, sister and their families for always cheering me on.

# Curriculum Vitae

## Samar Abdi

- 1998 B.Tech. in Computer Science and Engineering,  
Department of Computer Science and Engineering,  
Indian Institute of Technology, Kharagpur, India
- 1998-2000 Member of Technical Staff,  
Cadence Design Systems, Noida, India
- 2000-2001 Graduate Research Assistant,  
School of Information and Computer Science,  
University of California, Irvine, USA
- 2001-2002 Teaching Assistant,  
School of Information and Computer Science,  
University of California, Irvine, USA
- 2002-2005 Graduate Research Assistant,  
School of Information and Computer Science,  
University of California, Irvine, USA
- 2003 M.S. in Information and Computer Science,  
School of Information and Computer Science,  
University of California, Irvine, USA
- 2005 Ph.D. in Information and Computer Science,  
University of California, Irvine, USA  
Dissertation: *Functional Verification of System Level Model Refine-  
ments*

## Publications

J. Peng, S. Abdi, D. Gajski, “Model Refinement for Fast Architecture Exploration”, Joint Asia and South Pacific Design Automation Conference (ASP-DAC) and VLSI Design Conference, Bangalore, India, 2002.

S. Abdi, D. Shin, D. Gajski, “Automatic Communication Refinement for System Level Design”, Design Automation Conference (DAC), Anaheim, CA, USA, June 2003.

D. Shin, S. Abdi, D. Gajski, “Automatic Generation of Bus Functional Models from Transaction Level Models”, Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, Japan, January 2004.

S. Abdi, D. Gajski, “On Deriving Equivalent Architecture Model from System Specification”, Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, Japan, January 2004.

S. Abdi, D. Gajski, “Automatic Generation of Equivalent Architecture Model from Functional Specification”, Design Automation Conference (DAC), San Diego, CA, USA, June 2004.

S. Abdi, D. Gajski, “Model Validation for Mapping Specification Behaviors to Processing Elements”, Workshop on High Level Design Validation and Test (HLDVT) , Sonoma, CA, USA, November 2004.

J. Peng, S. Abdi, D. Gajski, “A Clustering Algorithm for Optimization of HW/SW Synchronization”, Asia and South Pacific Design Automation Conference (ASP-DAC) , Shanghai, China, January 2005.

S. Abdi, D. Gajski, “A Formalism for Functionality Preserving System Level Transformations”, Asia and South Pacific Design Automation Conference (ASP-DAC) , Shanghai, China, January 2005.

S. Abdi, D. Gajski, “Functional Validation of System Level Static Scheduling”, Design Automation and Test in Europe (DATE) , Munich, Germany, March 2005.

# Abstract of the Dissertation

Functional Verification of System Level Model Refinements

by

Samar Abdi

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2005

Professor Daniel D. Gajski, Chair

With continuous improvement in process technology, designers have more resources than ever available for implementing their systems. As applications become larger, the modeling abstraction has to rise to keep the complexity manageable. This has given rise to modeling at abstractions above the register transfer and cycle accurate levels. Verification becomes an even greater challenge as designers try to develop more models to evaluate their implementation choices. In this dissertation we present our verification strategy to alleviate the problems resulting from the move to system level. We argue our case for a system level modeling and verification methodology where detailed models are refined from abstract models. The models are represented formally using Model Algebra and the refinements are proven to be functionality preserving, using the rules of Model Algebra.

We define the objects and composition rules of Model Algebra and show how system level models are represented as expressions in this formalism. The formal execution semantics of such models are discussed and a notion for functional equivalence of models is also presented. We then define transformation rules that syntactically modify a model algebraic expression, while

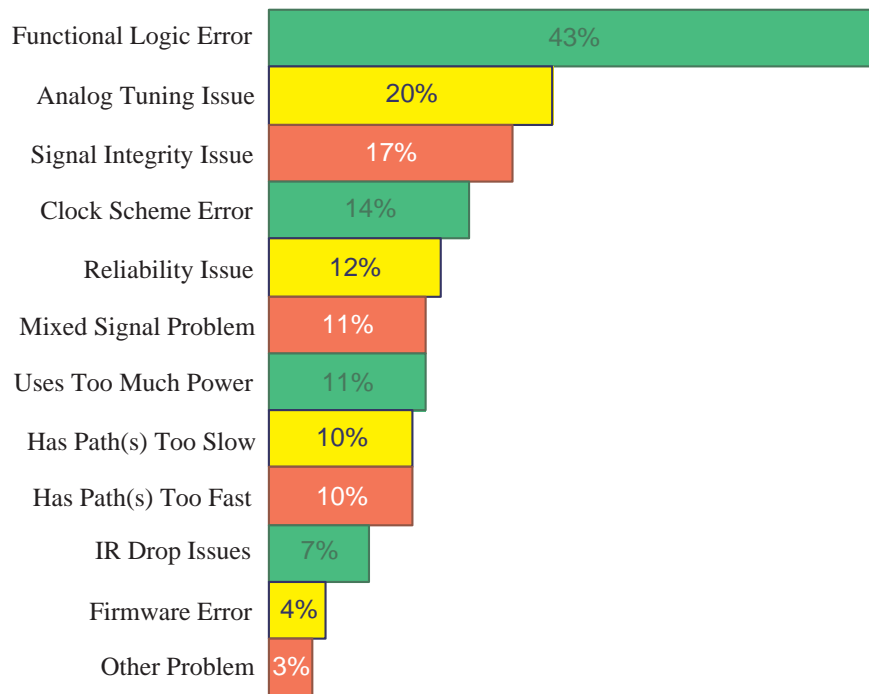
keeping its functionality intact. Then we present key refinements that are commonly encountered in system level design and prove the correctness of those refinements using the transformation rules of Model Algebra. As a proof of concept, we implemented a tool that automatically checks if the refinements were performed correctly on SpecC models. As a result, the designer needs to perform the costly and time consuming property verification on the first model only. All step wise refinements from the original model are proven to produce functionally equivalent models.

# Chapter 1

## Introduction

The continuous increase in size and complexity of System-on-Chip designs has introduced new modeling and verification challenges. Figure 1.1 shows a survey results for the reasons behind first time chip respins. It was reported that 61% of all chips designed in 2003 required at least one respin. With the spiralling cost of masks and short time to market, every respin sets back design companies not only in design costs but also in lost market penetration opportunity. The clearly leading reason for respins was functional verification amounting up to half of all respins.

Traditional design automation tools work with models at the cycle accurate level or below. As applications become larger, the lower level details like cycle accurate timing or implementation of protocols must be abstracted away to get a better understanding of the design and for ease of verification. The adoption of the system level (SL) of design abstraction is breeding a new generation of languages and tools [GZD<sup>+</sup>00, Sysa] for verification of abstract models. Most of this verification is simulation based. Since SL models simulate several orders of magnitude faster than their pin and cycle accurate counterparts, system designers choose to develop several SL models to evaluate possible implementation alternatives. Typically, such SL models are refined from a golden reference model. However, without an underlying formalism and well defined semantics for SL models, we will inevitably run into the problem of having too many models for the same design that cannot be synthesized or verified for functional equivalence. This dissertation attempts to alleviate this problem by introducing a formalism for representation of SL model, and using that formalism to develop reliable algorithms for refining one SL model into another.



Source: Aart de Geus, Chairman & CEO of Synopsys, SNUG keynote 2003.

Available: <http://www.deepchip.com/posts/0417.html>

Figure 1.1: 2003 survey results for reasons behind chip respins

In this introductory chapter, we present current system modeling styles and verification methodologies. We also propose our modeling and verification methodology that is based on correct step wise refinement of abstract SL models into more detailed models.

## 1.1 Traditional System Verification

Systems consisting of both HW and SW components are typically specified at the high level using a C description or a MATLAB [Mat] model that serves as the golden reference for design of both HW and SW. The SW application code is typically written in a high level language like C. The HW application is converted to a cycle accurate model in a hardware description language(HDL), either manually or using behavioral synthesis tools [For]. The glue logic, consisting of the HW interfaces on one side and the processor core (along with parts of the RTOS) on the SW side, allows co-simulation of the entire systems at cycle accurate detail. However, the simulation

speed of such models is painfully slow since there are two different simulation kernels for HW and SW. SL design languages like SystemC and SpecC ease this problem by providing a single language for expressing both HW and SW and a single simulation kernel. They also allow connection elements called channels that allows communication between different components using function calls instead of pin-level wire wiggling. Thus, system simulation becomes dramatically faster. However, the accuracy of protocol details and timing is lost.

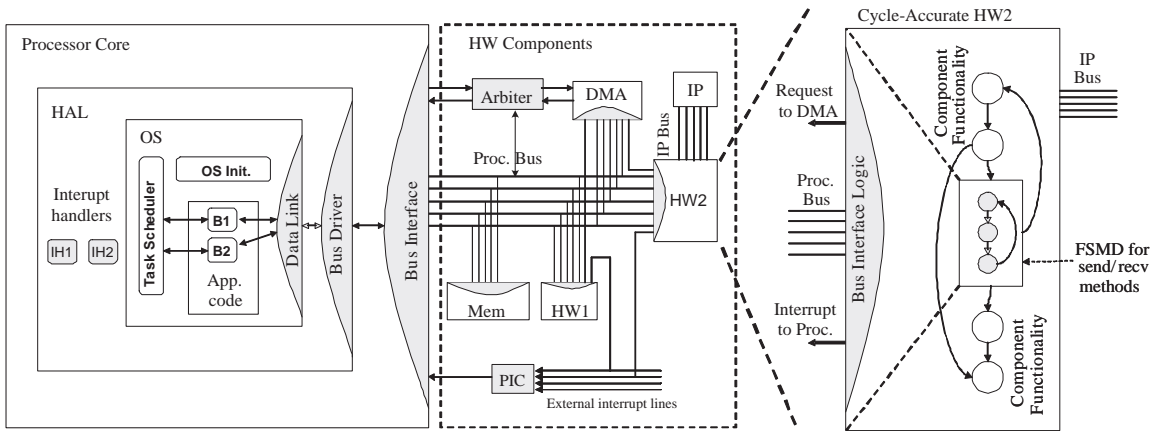


Figure 1.2: Pin-Accurate System Model

### 1.1.1 Pin and Cycle Accurate Simulation

Figure 1.2 shows the pin-accurate model of the system. A complex system may consist of one or more processor and a bunch of peripherals like DSPs, custom hardware units, memories, IPs etc. These components are connected to each other via ports on their interface. In order to implement communication between these components, we need to implement additional communication elements such as bus arbiters and programmable interrupt controllers (PICs). These elements form part of the communication sub-system along with the bus drivers implemented inside each component.

On the SW side, the processor core consists of a bus interface to the processor bus. Special HW registers are assigned to control bit toggling on the bus wires to enable communication. The system software consists of a hardware abstraction layer (HAL) that implements the bus drivers, responsible for sending and receiving bus words using the underlying interface logic.

These drivers implement the cycle accurate bus protocol, as described in the data sheet of the processor. The OS layer provides higher layers of communication abstraction, that hide the actual communication events from the application. Thus, the application may be able to perform message level transactions using the underlying services.

On the HW side, the communication primitives are implemented as cycle accurate HDL code, which may be inlined into the rest of the HW functionality. The FSM used for communication implements primitives like sending interrupts and following the detailed bus protocol. Finally, memory devices implement controllers that follow the memory access protocol. We will consider all memory components to be slaves that only provide read and write service on the bus. As mentioned earlier, the complete system may be modeled in system design languages. Alternately, we can model the SW part on the ISS, which will give us accurate performance numbers since the ISS model of the processor includes internal details such as pipelining etc.

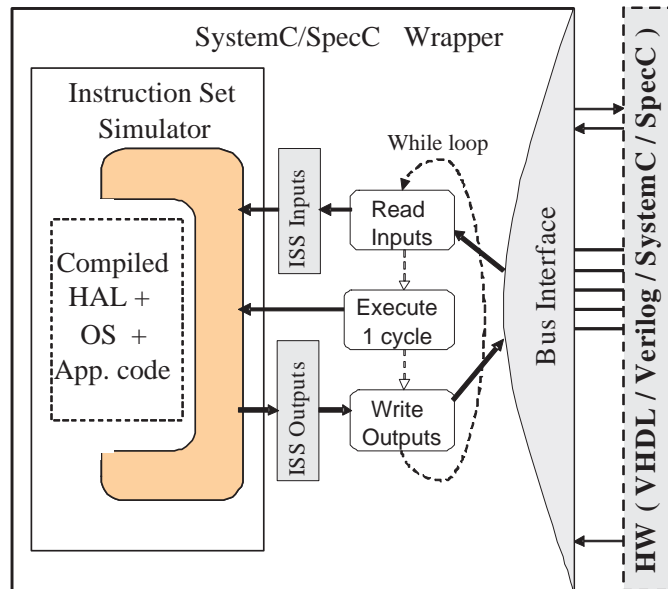


Figure 1.3: Cycle accurate simulation environment

Figure 1.3 shows the simulation environment for the pin-accurate model described above. The application code, along with the system software like the HAL and OS are compiled into a single binary. This binary is then loaded into the ISS of the processor. On its interface, the ISS model provides ports identical to those on the processor core. For simulation purposes, these ports are connected to the signals that communicate with rest of the system.

The ISS is usually a C model that must be connected to an HDL or SystemC simulator for complete system simulation. HDL simulators like those for Verilog and VHDL come with standard C interfaces into which these ISS models can be plugged. Figure 1.3 shows how the ISS can be wrapped inside a SystemC or SpecC wrapper for co-simulation with rest of the system. At every simulation cycle, the inputs to the processor interface are read and provided to the ISS. Then the cycle is advanced and the “Execute 1 cycle” method of the ISS is invoked from the wrapper. After the cycle, the ISS updates the output ports of the processor model, which are sampled by the wrapper. The remaining system, consisting of HW components or other processors executes in parallel to the ISS.

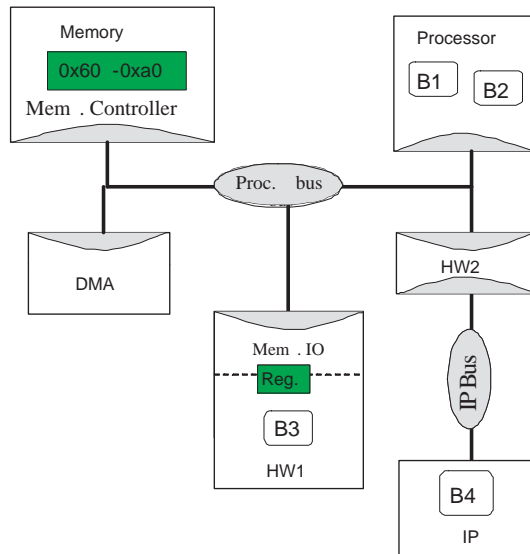


Figure 1.4: A transaction level model of the system

### 1.1.2 Transaction Level Simulation

It can be seen that verification at pin and cycle accurate level can be very slow not only due to two different simulation kernels, but also because every bus transaction wiggles several bus wires that generate too many events. All these events have to be processed by the simulator that significantly slows down model execution. One way to abstract away the pin-level simulation details is to implement each transaction as a function call that uses events to synchronize and word level assignments for data transfer. This is the basic concept behind transaction level model

as shownn Figure 1.4, where the system connectives are channels (shown by ovals) that provide services like synchronization and data transfer in the form of functions.

## 1.2 Proposed System Verification

Our verification technique works in a system level design methodology where increasingly detailed models are refined from high level models. We define clear semantics of models and design decisions that are need to refine models correctly.

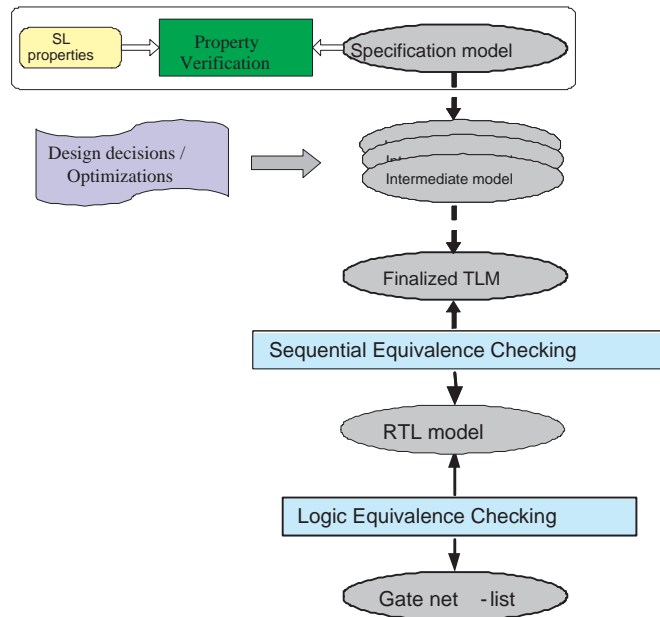


Figure 1.5: An illustration of proposed verification methodology

### 1.2.1 System Design Methodology

Figure 1.5 shows our system design and verification methodology. We start with a well defined executable specification model that serves as the golden reference. Before settling on the specification it must be thoroughly verified using simulation or formal methods against a set of system properties. Then the specification is gradually refined to a transaction level model that forms the basis for developing pin and cycle accurate models. The latter can be fed to traditional design automation tools. Transaction level models and RTL models can be compared for func-

tional equivalence using sequential equivalence checking [SDN87, sle]. Similarly, RTL models and gate level models can be compared for equivalence using combinational logic equivalence checking [GPB01]. Both these verification techniques are viable due to the used modeling formalisms, namely finite state machines and boolean algebra. However, no such universally adopted formalism exists above the cycle accurate level.

The gradual refinement produces intermediate models depends on the design decisions. If the model semantics and underlying formalism are not well defined, the refinement process cannot be automated or verified for functional correctness. In our methodology, each refinement translates to several small model transformations. Model semantics at each abstraction level and all the transformation rules are formally defined and proved for correctness. Therefore the refinement process can be reliably automated.

### **1.2.2 Formal Modeling**

Formally, a model is a set of objects and composition rules defined on the objects [AG04c]. A system level model would have objects like behaviors for computation and channels for communication. The behaviors can be composed as per their ordering. The composition creates hierarchical behaviors that can be further composed. Interfaces between behaviors and channels or amongst behaviors themselves can be visualized as relations. This allows us the ability to create complex models as a hierarchy of small computation objects linked with communication objects.

### **1.2.3 Verification by Correct Refinement**

A transformation on a model can be expressed using the concept of rearranging and replacing objects. For instance, in order to distribute the behaviors in a specification onto components of the system architecture, we need to rearrange the behaviors into a different hierarchical composition, based on the distribution. In order to use IP components, we need to replace behaviors in the model with an IP from the library. Each of these transformations has to be proven correct in a formal context.

A model refinement can be expressed as a well defined sequence of transformations.

Therefore, a refinement, say  $R$  of model  $M$  is a syntactic manipulation of  $M$  resulting in a new model  $M'$ . Refinement  $R$  can be written as a sequence of transformations  $t_1$  through  $t_n$  in that order. Hence, we can write

$$M' = R(M) = t_n(t_{n-1}(\dots t_1(M)\dots))$$

Since each transformation is shown to be correct, the refinement also produces an output model equivalent to the input model, by transitivity. A refinement based methodology can be defined as a set of well defined models and the refinements between them.

The notion of model equivalence comes from the simulation semantics of the model. Two models are equivalent if they have the same simulation results. This translates to the same (or equivalent) objects in both models and the same partial order of execution between them. Correct refinement, however, does not mean that the output model is bug free. We also need to use traditional verification techniques on the specification model and prove equivalence of objects that can be replaced with each other.

### 1.3 Thesis organization

The rest of this dissertation is organized as follows. In Chapter 2, we present an overview of existing methods in verification of embedded systems. We will present both simulation based and formal verification techniques and give arguments on the suitability of these approaches in different design scenarios. In Chapter 3, we present a formalism, called Model Algebra (MA), that is used to represent system level models and reason about their functional equivalence. We define the objects and composition rules of MA and show how system level models can be represented as expressions in this formalism. We also define the formal execution semantics of models written in MA and discuss notion of equivalence for such models. Then we present functionality preserving transformation rules on such models. Chapter 4 deals with key system level model refinements and how their correctness is proved using the transformation rules. The implementation of an automatic tool for verifying such refinements is presented in Chapter 5. Finally, we wind up with conclusions in Chapter 6 by outlining, the benefits of our verification approach, the contributions of this thesis and outlook for future research in this direction.

## Chapter 2

# System Verification Techniques

Verification is the most time consuming stage of system design. By most estimates, more than 70% of design effort is spent in verifying if the design meets the specification requirements. Most of this effort is directed in functional or logic verification. The other, equally important, facet of verification is performance verification, where the designer tries to check if the final manufactured device will meet the specified constraints of timing, area and power, among other metrics. In this chapter, we will give a brief overview of current industrial practice of design verification and the underlying techniques.

The verification methods available today can be broadly classified into two categories, namely simulation based and formal methods. In simulation based methods, the designer writes an executable model of the design. Test vectors are applied to the inputs of the model and output values are generated after delays as specified in the model. The functionality of the model is tested by comparing the generated outputs to the expected outputs. Pure formal methods do not need a simulation environment. The models and properties are expressed in a mathematical form and mathematical formulations are used to either compare two models or check if a property holds in a model. Semi-formal methods primarily use a simulation environment, but apply symbolic methods for stimulating and monitoring the design. The gain is in the absence of test cases, however monitoring simulation results becomes more complicated. This is because the monitor has to compare generated output expressions against expected output expressions using formal methods.

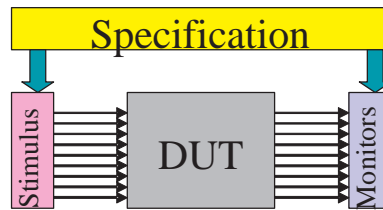


Figure 2.1: A typical simulation environment

## 2.1 Simulation based Methods

Simulation is the most widely used method for validation of models. A typical simulation environment is shown in Figure 2.1. The design to be tested is described in some modeling language and is referred to as design under test (DUT). The design specification is then used to generate input and output test vectors. The stimulus routine applies the input vectors to the models. The inputs are propagated through the model by the simulation tool and finally the outputs are generated. A monitor routine checks the output of the DUT against expected outputs for each input test vector. If a mismatch is found, the designer can use debugging tools to trace back and find the source of the problem. The problem arises from either incorrect design or incorrect timing. Once the problem source is identified, the designer can fix it and simulate the new model.

The intent of the designer is to test the model for all possible scenarios. However, this would require an unreasonable number of test vectors. Since only a limited number of test vectors will be used, the designer must try to choose the most useful ones. The usefulness of a test case is usually defined by the number of components and connections it can cover. Moreover, a test case that verifies an already tested part of the design does not add any value. Therefore, several coverage metrics have been invented to quantify the usefulness of a test case.

The simulation performance can be improved either by speeding up the simulator or by choosing test cases intelligently to maximize coverage with minimal simulation runs. One optimization is to reduce test generation time by giving constraints to stimuli and testing with only valid inputs. Monitoring non-primary output variables in the model reduces debug time by pointing out the error closer to its source. Testing the model by prototyping it on hardware provides much faster functional testing. Finally, rewriting the model by abstracting away low-level details also reduces simulation time, thereby finding functional errors earlier.

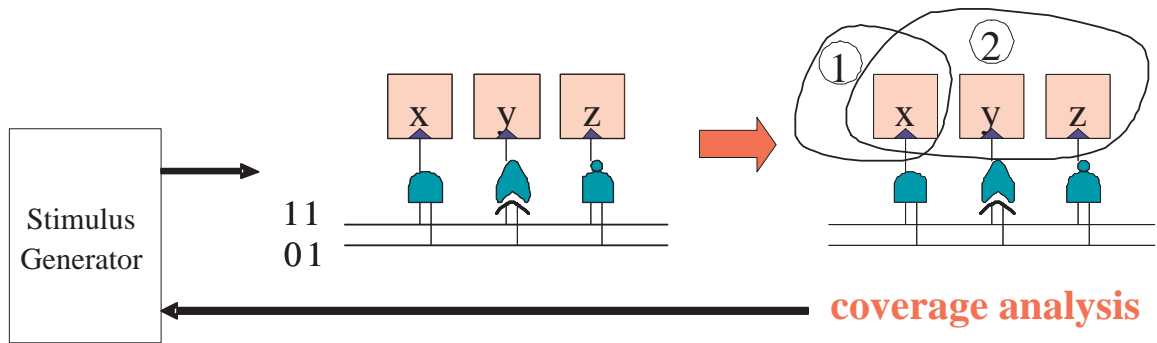


Figure 2.2: Simulation optimization using coverage feedback

### 2.1.1 Stimulus Optimization

Writing down test vectors for simulation can be a painful task. Also, generating test vectors randomly might result in a lot of invalid vectors. Since the model is typically constrained to work for only select scenarios, we can use this knowledge to generate valid test vectors only. The test scenario can thus be written in some language like "e" [eve] and SystemVerilog [Sysb] and a tool can be used to generate valid test vectors for that scenario.

Using the results from coverage is another way to minimize the number of test vectors. For instance, the code coverage feedback technique can be visualized in Figure 2.2. A simulation run with vector "11" results in only block "x" being covered. The designer looks at the coverage result and comes up with a vector "10" to cover blocks "y" and "z". Note that vector "00" would not cover block "y" and is thus not used. Such a feedback strategy can be used with other coverage metrics as well.

An increasingly popular method to minimize the effort in generating test vectors is by using explicit properties in the model. These properties, called assertions, are typically temporal in nature and may be embedded in the model either as special comments (pragmas) or in a different language than the modeling language. Some languages like System Verilog provide special syntax for specifying assertions. The assertions are parsed by a special test vector generator that automatically creates constrained test vectors, thereby freeing the designer from writing explicit tests for all manifestations of the assertion.

### 2.1.2 Monitor Optimization

Monitoring only the primary outputs of a design during simulation lets us know if a bug exists. Tracing the bug to its source can be difficult for a complex design. If the source code of the model is available, assertions can be placed on internal variables or signals in the model. For example, we can specify that the two complementary outputs of a flip-flop never evaluate to the same value. Not only does this improve understanding of the design, it also points out the bug much closer to the source. Assertions can also be used to check validity of properties over time, like protocol compliance.

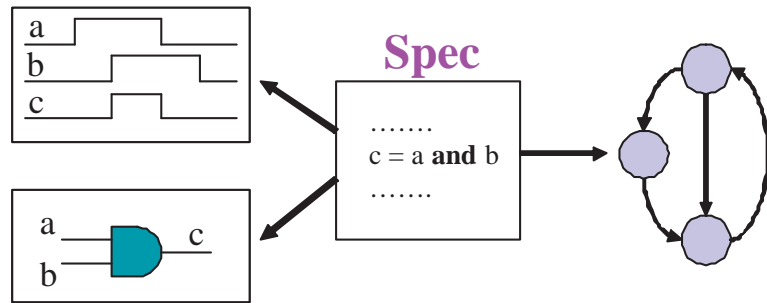


Figure 2.3: Graphical visualization of model execution

Graphical visualization of the structure and behavior of a design also helps debugging. Specifically, correlation between different representations, such as waveforms, net lists, state machines and code, as shown in Figure 2.3, allows the designer to easily identify the bug in a graphical representations and locate the source code for buggy part of the model.

### 2.1.3 Speed up techniques

Overall simulation time can be reduced by simply increasing the simulation speed. The two common speedup techniques are cycle simulation and emulation. Cycle simulation is used when we are only concerned about the signals at clock boundaries. This allows improving the simulation algorithm to update signal values at clock boundaries only. On the other hand event driven simulation needs to keep track of all events, even between the clock edges, and is thus much slower.

Another speedup technique is the use of reconfigurable hardware to implement the DUT.

If the designer wants to simulate a component in a larger available system, the FPGA implementation can be hardwired in the system. This technique is called in-circuit emulation [Gan01]. A different scenario in which emulation is used is dubbed simulation acceleration [SBR05]. The synthesizable part of the hardware is implemented on an FPGA. The SW and the unsynthesizable HW runs on a software simulator, which talks to the emulation tool via remote procedure calls. It must be noted that FPGA prototyping is meant for functional verification only. Synthesis results for ASICs and FPGA implementation of the same design can be very different in timing, area and power. Therefore, if the design is to be finally implemented in ASIC, the FPGA prototype only provides close to “at-speed” functional verification.

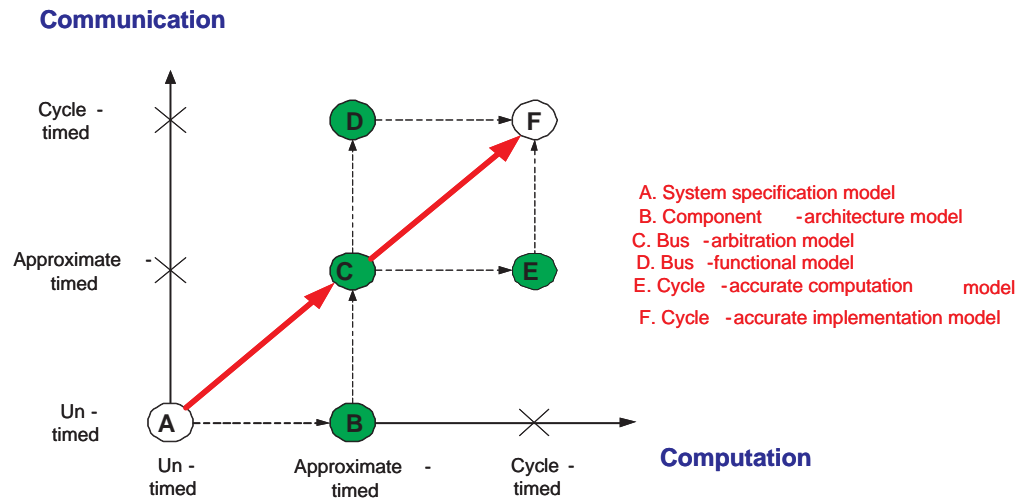


Figure 2.4: System level models with different computation and communication details

### 2.1.4 Modeling Techniques

A different approach to reduce functional verification time is by modeling the system at higher abstraction levels. By abstracting away unnecessary implementation details, the model not only becomes more understandable, but also simulates faster. For instance, models with bus transactions at word level simulate faster than those at bit level because the simulator does not have to keep track of bit-toggling on bus wires. Similarly, models with coarse timing result in fewer events during simulation. There are several abstract models that can be used depending on the size and nature of the design as well as the design methodology. Figure 2.4 illustrates system level

models at different levels of abstraction by placing them relative to the two axes of computation and communication detail [CG03, Don04].

## 2.2 Formal Verification Methods

Formal verification techniques use mathematical formulations to verify designs. In this section, we will discuss three basic techniques of formal verification, namely equivalence checking, model checking and theorem proving. In order to check for correctness of synthesis and optimization of models, we can use equivalence checking. We define some notion of equivalence like logic equivalence or state machine equivalence and the equivalence checker proves or disproves the equivalence of original and optimized/synthesized models. Model checking, on the other hand, takes a formal representation of both the model and a given property, and checks if the property is satisfied by the model. Theorem proving takes formal representations of both the specification and implementation in a mathematical logic and proves their equivalence using the axioms and inference rules of the logic.

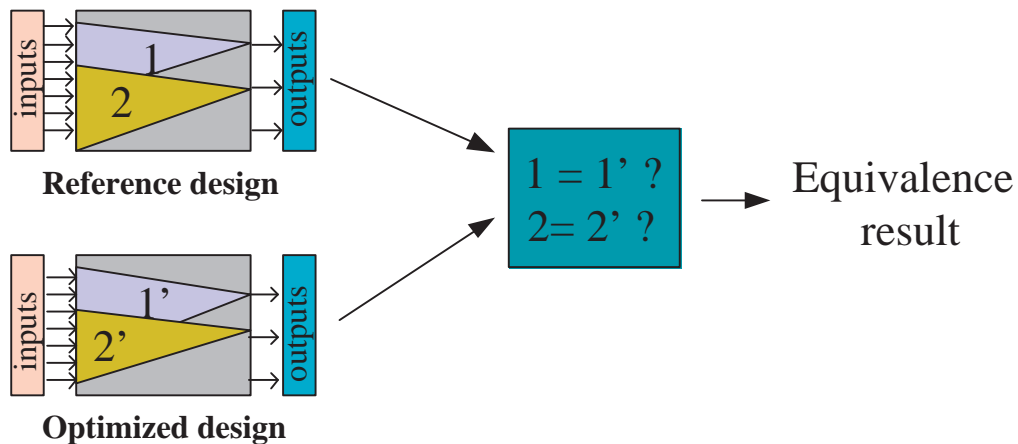


Figure 2.5: Logic equivalence checking

### 2.2.1 Logic Equivalence Checking

During synthesis or optimization of logic circuits, the design is optimized to reduce the number of gates or circuit delay. The designer is responsible for the logical correctness of

any such transformation. A logic equivalence checker checks that the result of the synthesis or optimization is equivalent to the original design. This is achieved by dividing the model into logic cones between registers, latches or black-boxes as shown in Figure 2.5. The corresponding logic cones are then compared between original and optimized models.

Logic cones can be described with boolean expressions and thus represented as boolean decision diagrams (BDDs) [Bry86] or some normal representation like conjunctive normal form (CNF). Since BDDs have a canonical form, we can reduce the original and optimized cones to their respective canonical forms and check if they are identical. This requires an ordering of the boolean variables in the expression. The size of the BDD is highly sensitive to this ordering. Alternately, satisfiability solvers can be used to compare the boolean expressions. This technique is the most popular formal technique in industry and is widely applied at the GATE/RTL level to ensure correctness of logic synthesis.

### 2.2.2 FSM Equivalence Checking

Logic equivalence checker checks only the equivalence of the combinational part of the circuit. There are also techniques to check equivalence of the sequential part of the design [SDN87]. In order to understand those techniques, we have to define the notion of a finite state machine. A finite state machine (FSM) is a tuple consisting of a set of inputs, a set of outputs and a set of states. Some of the states are designated as initial states and some as final states. Transitions between states are defined as a function of current state and the input. An output is also associated with every state. Formally, we can write

$$M = \langle Q, I, O, S, \mathcal{F}, \delta, \sigma \rangle, \text{ where}$$

$Q$  = set of all states

$I$  = set of all inputs

$O$  = set of all outputs

$S \subset Q$  = set of initial states

$\mathcal{F} \subset Q$  = set of final states

$\delta \subset Q \times I \times Q$  = transition function

$\sigma \subset Q \times O$  = output labeling of each state.

We can think of a FSM as a language acceptor. Assume that we start from an initial state and supply input symbols from a string  $S$ , with each input causing a state transition. If we reach a final state once all the inputs from  $S$  are exhausted, then  $S$  is said to be accepted by the FSM. The set of all acceptable strings forms the language of the FSM. We also define the notion of a product FSM. The product of two finite state machines  $M_1$  and  $M_2$  has the same behavior as if  $M_1$  and  $M_2$  were running in parallel. Formally, given

$$M_1 = (Q_1, I_1, O_1, S_1, F_1, \delta_1, \sigma_1)$$

$$M_2 = (Q_2, I_2, O_2, S_2, F_2, \delta_2, \sigma_2)$$

The product state machine  $M = M_1 \times M_2$  is defined as

$$M = (Q, I, O, S, F, \delta, \sigma), \text{ where}$$

$$Q = Q_1 \times Q_2$$

$$I = I_1 \times I_2$$

$$O = O_1 \times O_2$$

$$S = S_1 \times S_2$$

$$F = F_1 \times F_2$$

The transition function is defined as follows. If there is a transition from state  $q_1$  to  $q'_1$  for input  $i_1$  in  $M_1$  and there is a transition from state  $q_2$  to  $q'_2$  for input  $i_2$  in  $M_2$ , then there is a transition from  $(q_1, q_2)$  to  $(q'_1, q'_2)$  for input  $i_1, i_2$  in  $M_1 \times M_2$ . The output labeling of  $M_1 \times M_2$  is defined as follows. If  $(q_1, o_1) \in \sigma_1$  and  $(q_2, o_2) \in \sigma_2$ , then  $((q_1, q_2), (o_1, o_2)) \in \sigma$ .

We can now define equivalence of FSM models by using the previously discussed notions and concepts. The specification and its implementation are both represented as FSMs  $M_s$  and  $M_i$  respectively. It must be ensured that the input and output alphabet of the two machines should be the same.

We derive the product machine  $M_s \times M_i$  using cross product of FSMs. Now all the states in  $M_s \times M_i$  that have pair of differing outputs are labeled as final states. In Figure 2.6, the states  $ps$ ,  $pt$  and  $qr$  have output pairs with non-identical symbols ( $xy$  or  $yx$ ) and are thus labeled as final states. We also keep only those transitions that have the same symbols in the input pair. What we are trying to prove is that for the same sequence of inputs,  $M_s$  and  $M_i$  would produce the

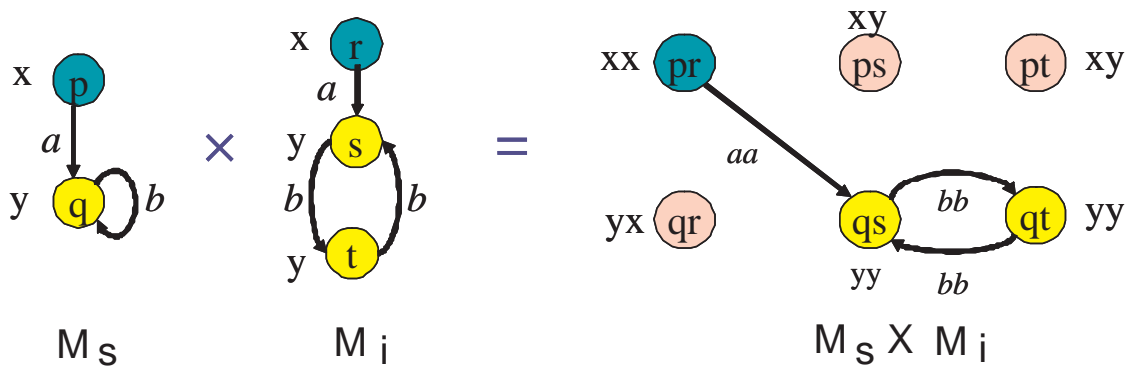


Figure 2.6: FSM equivalence checking

same sequence of outputs. In other words, any state with a pair of non-identical outputs should never be reached. Since such states are the final states in the product FSM, they should never be reached. Therefore the product FSM should not accept any language. This notion is called language emptiness. Showing language emptiness means starting from the set of initial states in  $M_s \times M_i$  and performing a reachability analysis. If any of the final states is reachable, then the specification and implementation are not equivalent.

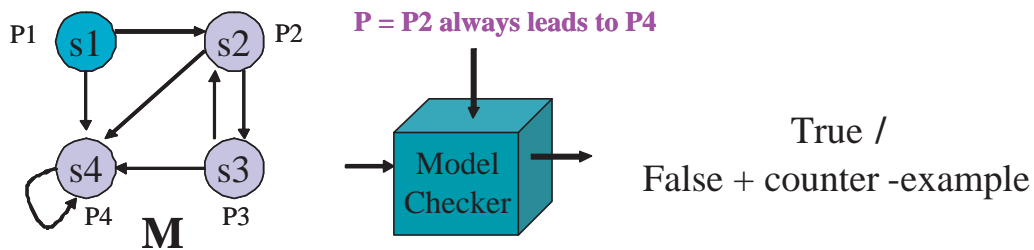


Figure 2.7: Model checking of a transition system

### 2.2.3 Model Checking

Model checking [CGP00] is a formal technique for property verification illustrated in Figure 2.7. The model is represented as a state transition system (also called a Kripke structure), which consists of a finite set of states, transitions between states and labels on each state. The state labels are atomic properties that hold true in that state. The model checking problem is simply defined as a verification that a temporal property  $P$  is satisfied by the model defined as a state

transition system  $(M,s)$ , where  $s$  is the start state for the model execution. We write the model checking problem as  $M,s \models P$ . For example, in the model of a D-flip flop the state variables would be the input, the clock, the output, its complement, and the reset. The states would be all possible values of the state variables. A simple property might be that if the reset signal is 0, then eventually the output will be 0. In figure 2.7 we loosely define  $P$  as “if  $P_2$  is satisfied, then  $P_4$  will always be satisfied”. The model checker needs to get such properties in a formal input format. Temporal logics [Pnu77] are such formalisms that may be used for specifying temporal properties.

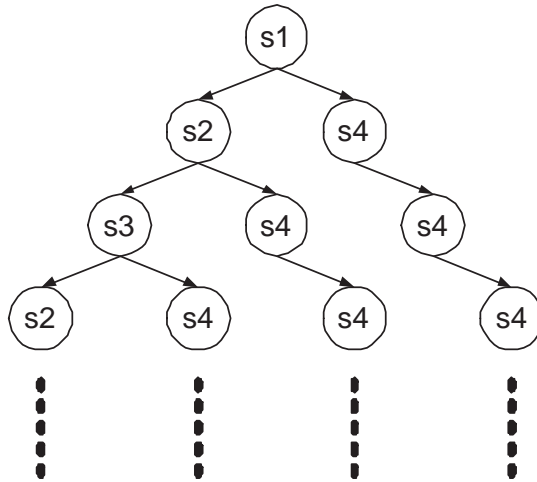


Figure 2.8: Example of a computation tree

The most common temporal logic is the computation tree logic (CTL\*). Formulas of CTL\* describe properties on computation trees. These trees are formed by taking a start state of the transition system as root and unrolling all reachable paths from that state. Temporal properties can be graphically visualized on this computation tree. Figure 2.8 shows the computation tree rooted at  $S_1$  for the transition system in Figure 2.7. CTL\* formulas consist of atomic properties, temporal operators and path quantifiers. Temporal operators include

- Next state operator  $X$ , where  $X(P)$  means that  $P$  will hold in the second state,
- Eventual operator  $F$ , where  $F(P)$  means that  $P$  will hold some time in the future,
- Always operator  $G$ , where  $G(P)$  means that  $G$  will hold all the time,
- Until operator  $U$ , where  $U(P_1, P_2)$  means that  $P_1$  will hold until  $P_2$  becomes true, but may not afterwards

- Release operator R, where  $R(P1, P2)$  means that P1 will start holding true from the last state that P2 was true

Besides the temporal operators, we define path quantifiers on the computation tree. Quantifier A(P) implies that P will hold true on all paths on the tree and quantifier E(P) implies that P will hold true for at least one path of computation tree. Using these operators and quantifiers, we can express our original property “if P2 holds then P4 will always hold” from figure 2.7. Since P2 only holds in state  $s_2$ , we can pose this model checking problem as

$$M, s_2 \models A(F(P4))$$

The model checker works on the state transition system of the model and the given property and produces a result TRUE if the property holds in the model. If the property does not hold, the checker gives a counter-example to show that the property is violated. This feature of model checking is very helpful in debugging because it provides a readymade test case.

The major problem with model checking is the state space explosion problem. The state transition system grows exponentially with the number of state variables. Therefore, memory for storing the state transition system becomes insufficient as the design size grows. Symbolic model checking [JEK<sup>+</sup>90] avoids creation of explicit graph data structures for representing state transition systems. Instead, the atomic properties and state transitions are expressed as a boolean expression of the state variables in the model. This alleviates the problem of too many states to a large extent.

## 2.2.4 Theorem Proving

In general, theorem proving [CRSS94, M.J88] is a technique where a model is specified in a formalism. The formalism provides for objects and composition rules to create models and a set of axioms and inference rules to apply transformations on a given model. Logic equivalence checking may be considered a special case of theorem proving, where the underlying formalism is boolean algebra.

Due to its generality and mathematical basis, theorem proving can be applied to almost any verification problem in any domain. In the domain of circuits and systems, higher order logic

is used to prove properties on models. The theorem proving problem may be posed as follows. Given two expressions in some logic, derive a proof using the rules of that logic and the problem domain for the expressions' equivalence or otherwise.

The proof uses certain assumptions about the problem domain and axioms of the mathematical logic. In the domain of circuit design, an assumption might be that power supply is always at logic level 1 while ground is logic 0. The proof is constructed by breaking down a complex proof goal into smaller goals. The smaller goals are then simplified using assumptions and then passed onto automatic theorem prover. Theorem proving is still a largely manual process. Several steps of manually simplifying and breaking down proof goals may be required before an automatic prover can solve it.

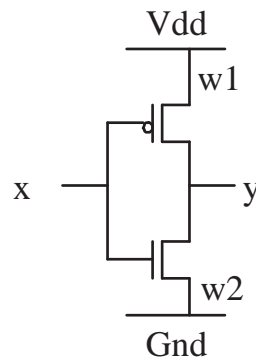


Figure 2.9: A CMOS inverter implementing  $y = \neg x$

We present a simple example of the use of theorem proving for verifying circuits due to Gordon [Gor88]. Suppose we have to prove that the CMOS inverter circuit, shown in Figure 2.9, inverts the input logic. We start with the basic assumptions about voltage levels and logic levels and the behavior of P and N transistors. The formula for the implementation is derived by conjunction of the various components of the inverter. The specification formula simply states that the output is logical inverse of input. The proof process takes the implementation formula and reduces it to the specification formula by a number of steps. Each proof step uses either an inference rule, an axiom or an already proven theorem.

For the CMOS inverter in Figure 2.9, let  $w_1$  be the net connected to power supply ( $VDD$ ). Let  $w_2$  be the net connected to ground ( $GND$ ). Let  $x$  be the input and  $y$  the output variable. Using the assumptions about PMOS and NMOS transistors, we get the following assumptions (or

inference rules)

$$\begin{aligned}
VDD(a) &\Rightarrow (a = T) \\
GND(a) &\Rightarrow (a = F) \\
NMOS(a, b, c) &\Rightarrow (a \rightarrow (b = c)) \\
PMOS(a, b, c) &\Rightarrow (\neg a \rightarrow (b = c))
\end{aligned} \tag{2.1}$$

The specification of the CMOS inverter may be written as

$$Spec(x, y) := (y = \neg x)$$

The implementation shown in the figure may be written as a structural composition of the basic modules defined in the assumptions 2.1. Hence, we have

$$Impl(x, y) := \exists w_1 w_2. VDD(w_1) \wedge PMOS(x, w_1, y) \wedge NMOS(x, w_2, y) \wedge GND(w_2)$$

We have to prove that the implementation meets the specification, or

$$Impl(x, y) = Spec(x, y) \tag{2.2}$$

We perform a forward proof of theorem in 2.2 as follows

$$\begin{aligned}
Impl(x, y) &= \exists w_1 w_2. VDD(w_1) \wedge PMOS(x, w_1, y) \wedge NMOS(x, w_2, y) \wedge GND(w_2), \text{ by definition} \\
&= \exists w_1 w_2. (w_1 = T) \wedge PMOS(x, w_1, y) \wedge NMOS(x, w_2, y) \wedge (w_2 = F), \text{ using 2.1} \\
&= \exists w_1 w_2. (w_1 = T) \wedge PMOS(x, T, y) \wedge NMOS(x, F, y) \wedge (w_2 = F), \text{ by substitution} \\
&= (\exists w_1. w_1 = T) \wedge PMOS(x, T, y) \wedge NMOS(x, F, y) \wedge (\exists w_2. w_2 = F), \\
&\quad \text{using } \exists a. t_1 \wedge t_2 = (\exists a. t_1) \wedge t_2 \text{ if } a \text{ does not occur in } t_2 \\
&= PMOS(x, T, y) \wedge NMOS(x, F, y), \text{ since } \exists w_1. w_1 = T \text{ and } (\exists w_2. w_2 = F) \text{ hold} \\
&= (\neg x \rightarrow (T = y)) \wedge (x \rightarrow (y = F)), \text{ using 2.1} \\
&\Rightarrow Impl(T, y) = (y = F), Impl(F, y) = (y = T), \text{ analyzing all valuations of } x \\
&\Rightarrow Impl(x, y) = (y = \neg x), \text{ by all implementation valuations} \\
&\Rightarrow Impl(x, y) = Spec(x, y), \text{ by definition}
\end{aligned}$$

### 2.2.5 Drawbacks of Formal Verification

Formal verification methods have not been as well accepted in the industry as simulation based methods because of several drawbacks. Logical equivalence checking works only for combinational logic and FSM equivalence checking requires both specification and implementation machines to have the same set of inputs and outputs. Model checking, besides suffering from the state explosion problem, is not suitable for all types of designs. Since it needs a state transition system, it works best for control intensive designs like protocol compliance etc. Automatic theorem proving has not become very popular in the industry because of several reasons. The foremost reason is the amount of manual intervention required in running the theorem prover. Since different applications have different kinds of assumptions and proof strategies, it is infeasible for a theorem proving tool to generate the entire proof automatically. Secondly, most designers lack a background in mathematical logic. Therefore, it requires a huge investment and long training time for them to start using theorem proving efficiently.

### 2.2.6 Improvements to Formal Verification Methods

There have been several improvements to formal techniques, particularly in model checking. Symbolic model checking encodes the state transition system using BDDs, which is much more compact than exhaustively enumerating the states and transitions. Bounded model checking [CKY03, KCY03] is a variant of model checking that checks if a model satisfies a property on paths of length at most  $K$ . The number  $K$  is incremented until a bug is found or the problem becomes intractable. Also, for bounded model checking, the entire kripke structure of the model need not be created, which alleviates the problem of state explosion.

Partial order reduction techniques [ABH<sup>+</sup>97] are usually used in model checking of asynchronous systems, where concurrent tasks are interleaved rather than being executed simultaneously. It uses the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. Abstraction techniques [CGL94] are used to create smaller state transition graphs. The specified property is described using some state variables. The variables that do not influence the specified property are eliminated from the model, thereby preserving the property while reducing the model size.

### **2.2.7 Semi-Formal Methods: Symbolic Simulation**

The idea behind symbolic simulation is to significantly minimize the number of simulation test vectors, for the same coverage, by using symbols instead of explicit test vectors. In symbolic simulation [Bry90], the stimulus applies boolean variables as inputs to the simulation model. During simulation, the internal variables and outputs are computed as boolean expressions of the input variables. In order to check for correctness, the output expression is compared with the expected output expression for logic equivalence. BDDs can be used to store the boolean expressions. Since, BDDs of equivalent boolean expressions can be reduced to the same canonical form, the equivalence of specified output expression to simulated output expression can easily be checked. However, as the logic depth increases, the size of the BDDs increases exponentially. Therefore, for larger circuits, where the BDD size may blow up, SAT solvers [MMZ<sup>+</sup>01] may be used.

## **2.3 Evaluation Metrics for Verification Techniques**

In order to determine the most suitable verification method, one can define some metrics to evaluate them. The three most common metrics that we discuss here are coverage, cost and scalability. Coverage of a verification method determines how much of the design functionality has been tested. Cost includes the money spent on purchase of tools, hiring of experts and the training of users. Scalability of the technique shows if there are any limitations on the size or type of design that we are verifying.

Formal verification claims to provide complete coverage. However, the coverage is limited to the given property and the richness of the formalism that is used for model representation. For instance, model checking covers all possible states in the state transition representation of the model for a given property. Logic equivalence checking covers the combinational part of the model only. Nevertheless, the coverage of formal methods, if they are applicable, is significantly more than that of simulation based methods for the same run-time. Using assertions in the design can help make better test cases that exercise the assertions, thereby ensuring that the tests are useful and valid. Pseudo random testing, on the other hand, would generate a lot of test inputs that are invalid for the design, and hence wasted.

Cost and effort of a verification method influences the design phase in which it is used. For instance, the preliminary phase usually employs simulation to uncover most of the easy bugs. This is because most designers have experience with simulation tools and debuggers and it is thus cost effective. It is counter intuitive to employ expensive tools and expert verification engineers during early bug hunting. As the verification process continues and the design models stabilize, bugs become harder to find. In the latter phase, specialized and more expensive techniques like model checking or theorem proving may be used. Assertions are also used to generate more directed tests and to verify correctness on corner cases. The universal 80-20 rule applies to most design verification cases, that is, 20% of the total cost uncover 80% of the bugs and vice versa.

The performance of a verification method on different sizes and abstraction levels of models determines its scalability. Some methods like logic equivalence checking are limited to RTL models or below. Similarly, model checking is constrained by the number of state variables in the model. Compared to other techniques, simulation scales very well for all types of models. Almost any executable model at any level of abstraction can be simulated.

If we look at the trend in the acceptance of verification techniques in the industry, we find that methods with a severe drawback have been generally avoided. Model checking suffers from poor scalability and theorem proving is way too expensive, thereby making equivalence checking the most commonly used technique in the industry. Similarly, assertion based techniques may require extra cost but they are replacing pseudo random simulation because of their better coverage. Universal adoption of new verification and assertion languages is testimony to this fact.

## **2.4 Future of System Verification**

The new challenges to verification of systems comes from the growth in size and complexity of designs. Individually verified components do not work together due to interface issues. Also the sheer size of designs makes modeling and verification too expensive and time consuming.

To answer this challenge, we look towards a disciplined system design methodology. To design complex systems on chip, the level of model abstraction has been raised. If the semantics of system level models is well defined, then they can be formalized. Consequently, we can define transformations from models at one abstraction level to another. The transformations can be

proven to produce equivalence models. Thus, the traditional methods can still be used at higher levels of abstraction while correct transformations will over, time remove, the need to verify lower level models for functionality.

## **2.5 Chapter Summary**

In this chapter, we provided an overview of various verification strategies being employed in the industry today. We showed how coverage of traditional simulation is improved by using directed test vectors instead of random vectors. Better monitoring techniques such as graphical views of simulation results aid in understanding and debugging of designs. Emulation techniques provide “at-speed” simulation of the design by actually prototyping it in hardware. We also looked at formal verification methods that are slowly, yet surely, making inroads into the verification market. We presented the theoretical aspects of formal methods such as logic equivalence checking, model checking, theorem proving and symbolic simulation.

In conclusion, we found that as the size and complexity of system designs increase, traditional techniques might not be able to keep pace. A system design methodology with well defined model semantics may be a possible solution to the problem. Specifying the design at a higher level of abstraction would make traditional verification and debugging tractable because of smaller model size. Well defined model semantics would make it possible to define and prove correct transformations for automatic model refinement. Therefore, formalisms for representing and correctly transforming models would make complete system verification much faster.

## Chapter 3

# Model Algebra

A modeling formalism may be defined as a set of objects and composition rules that represent relationships between the objects. Our goal is to have a formalism that can allow the designer to express executable system models at different levels of abstraction. For instance, one should be able to express a model that shows only the functionality of the system using the objects and composition rules of the formalism. Also, one should be able to express models with structural details, using the same objects and composition rules. Given a model and its abstraction level, one should be able to identify the various structural artifacts within the model. Finally, a model expressed in such a formalism, should be executable so that it may be used to evaluate the design. The formalism must, therefore, have clear execution semantics.

### 3.1 Definition

A system can be viewed as a block of computation; with inputs and outputs for stimuli and response, respectively. This computation block is composed of smaller computation blocks that execute in a given order and communicate amongst themselves. Thus, for modeling purposes, it is imperative to have primitives for computation and communication. We will refer to the computation units as behaviors. A behavior has ports that allow it to be connected to other behaviors. The units of communication are variables and channels. These communication objects have different semantics. Variables allow a “read, compute and store” style of communication, while channels support a synchronized double handshake style of communication. Composition

rules are used to create an execution order of behaviors and to bind their ports to either variables or channels. A system is thus represented as a hierarchical behavior composed of sub-behaviors communicating via variables and channels.

### 3.1.1 Objects

The objects of MA can be defined as the tuple

$\langle \mathcal{B}, \mathcal{C}, \mathcal{V}, I, \mathcal{P}, \mathcal{A} \rangle$ , where

$\mathcal{B}$  is the set of behaviors

$\mathcal{C}$  is the set of channels

$\mathcal{V}$  is the set of variables

$I$  is the behavior interface

$\mathcal{P}$  is the set of behavior ports

$\mathcal{A}$  is the set of address labels

We also define a subset  $\mathcal{B}^I$  of  $\mathcal{B}$  representing the set of identity behaviors. Identity behaviors are those behaviors that, upon execution, produce an output that is identical to their input. In general, we will use the convention of naming identity behaviors as  $e$  followed by a subscript. Each of the variables in  $\mathcal{V}$  has *type* associated with it. We define  $Q$  to be the subset of  $\mathcal{V}$  such that all variables in  $Q$  are of type **boolean**.

#### 3.1.1.1 Ports

Each behavior has an associated object called its interface. The interface carries the ports of the behavior that are represented by their association to the behavior. Hence, to internal behaviors of a hierarchical behavior, the port is seen as  $I \langle p \rangle$ , where  $p \in \mathcal{P}$ . The port is treated like any other local variable except that we restrict operations on it, depending on its *direction*. Local behaviors can either write to a port, in which case it is known as the *out-port*, or they may read from the port, in which case it is called the *in-port*. If both read and write are allowed, the port is called *inout-port*. When the same port  $p$  is accessed from outside of behavior  $b$ , it is written as  $b \langle p \rangle$ .

### 3.1.1.2 Addressing

Behaviors communicate with each other using either memory or channels. Essentially, memory based communication follows the SW programming paradigm, where one behavior writes data into a variable through an out-port and another behavior reads it via an in-port. Behaviors executing concurrently use synchronized data transactions amongst themselves for communication. Channels serve as the media for such transactions. Each transaction uses an address to identify the sender and the receiver behaviors. The transactions can, thus, be visualized to take place over virtual links, that are labeled by distinct addresses. Each of the links is associated with a channel. Hence, such a link may be written as  $c < a >$ , where the link uses channel  $c$  and has the address  $a$ . Two transactions on a channel cannot share a link if they might take place simultaneously. In other words, all transactions on a single link must be totally ordered in time.

## 3.1.2 Composition Rules

Composition rules on the objects in MA are defined as relations in MA. These relations may contain two or more objects. Each composition rule creates a term, which may be further composed, in a particular format, to create hierarchical behaviors.

### 3.1.2.1 Control flow

A control flow composition ( $R_c$ ) determines the execution order of behaviors during model simulation. We write the relation as

$$q : b_1 \& b_2 \& \dots \& b_n \rightsquigarrow b$$

where  $\forall i, 1 \leq i \leq n, b_i \in \mathcal{B} \cup I, q \in Q$ . The composition rule implies that  $b$  executes after **all** the behaviors  $b_1$  through  $b_n$ , called predecessors in the relation, have completed **and**  $q$  evaluates to TRUE.  $R_c$  is said to *lead to*  $b$  under the condition  $q$ . It implies a synchronization where  $b$  must wait for all predecessors to complete. The degenerate case of the control flow relation is of the form  $q_1 : b_1 \rightsquigarrow b$ . Here, we only have a single predecessor, so  $b$  may start executing after  $b_1$  if  $q_1$  evaluates to TRUE, even if there are other control flow relations leading to  $b$ . A relation with a TRUE condition, eg.  $1 : b_1 \rightsquigarrow b_2$  will be shorthanded as  $b_1 \rightsquigarrow b_2$ .

### 3.1.2.2 Non-blocking write

This composition rule ( $R_{nw}$ ) is used to indicate that a behavior uses its out-port to write to a variable or an out-port of its parent behavior. In the case of a write to a data variable, we use the expression

$$b \langle p \rangle \rightarrow v$$

where  $b \langle p \rangle$  is the out-port of the writing behavior and  $v$  indicates the memory into which the data is written. In its other manifestation, this composition rule can be used to create a port connection, written as

$$b \langle p \rangle \rightarrow I \langle p' \rangle$$

In this case, the composition rule indicates a port-map in a hierarchical behavior. Note that  $\langle p' \rangle$  must also be an out-port or inout-port.

### 3.1.2.3 Non-blocking read

This composition rule ( $R_{nr}$ ) is used to indicate that a behavior uses its in-port to read data from a variable or through an in-port of its parent behavior. In the case of a read from a data variable, we use the expression

$$v \rightarrow b \langle p \rangle$$

where  $b \langle p \rangle$  is the in-port of the reading behavior and  $v$  indicates the memory from which the data is read. In its other manifestation, this composition rule can be used to create a port connection, written as

$$I \langle p' \rangle \rightarrow b \langle p \rangle$$

In this case, the composition rule indicates a port-map in a hierarchical behavior. Note that  $\langle p' \rangle$  must also be an in-port or inout-port.

### 3.1.2.4 Channel transaction

This composition rule ( $R_t$ ) indicates a data transfer link from the sender behavior to one or more receiver behavior(s) over a channel. The semantics of the composition rule ensure that the sender and the receiver(s) are ready at the time of the transaction. In other words, it follows a

rendezvous communication mechanism. The sender and receiver ports as well as the logical link of the channel are also indicated in the relation. We write this relation as

$$c \langle a \rangle : b \langle p \rangle \mapsto b_1 \langle p_1 \rangle \& b_2 \langle p_2 \rangle \dots \& b_n \langle p_n \rangle$$

where  $b \langle p \rangle$  is the out-port of the sending behavior and  $b_1 \langle p_1 \rangle$  through  $b_n \langle p_n \rangle$  are the in-ports of the receiving behaviors. The transaction takes place over channel  $c$  and uses the link addressed  $a$ .

### 3.1.2.5 Blocking write

This composition rule ( $R_{bw}$ ) is used to indicate the port connection for the sender part of a transaction. The sender behavior writes to the out-port of its parent behavior through one of its own out-ports. Eventually, the port will be bound to a channel transaction. Thus, the blocking write relation facilitates the creation of hierarchy in the model. We represent a blocking write by the expression

$$\langle a \rangle : b \langle p \rangle \mapsto I \langle p' \rangle$$

where  $b \langle p \rangle$  is the out-port of the writing behavior. The port  $I \langle p' \rangle$  on the parent behavior of  $b$  will eventually be bound to another blocking write relation or a channel transaction relation with address  $a$ .

### 3.1.2.6 Blocking read

This composition rule ( $R_{br}$ ) is used to indicate the port connection for the receiver part of a transaction link. The receiving behavior(s) read(s) from the in-port of their parent behavior through one of their own in-ports. Eventually, the port of the parent behavior will be bound to a channel transaction. Thus, the blocking read relation facilitates the creation of hierarchy in the model. We represent a blocking read by the expression

$$\langle a \rangle : I \langle p' \rangle \mapsto b_1 \langle p_1 \rangle \& b_2 \langle p_2 \rangle \dots \& b_n \langle p_n \rangle$$

where  $b_1 \langle p_1 \rangle$  through  $b_n \langle p_n \rangle$  are the in-port(s) of the receiving behavior(s). The port  $I \langle p' \rangle$  will eventually be bound to another blocking read relation or a channel transaction relation. The address of the virtual link ( $\langle a \rangle$ ) will be used for binding this port.

### 3.1.2.7 Grouping

This composition rule ( $R_g$ ) is used to indicate a collection of *terms* formed using above composition rules. Essentially, grouping is used to create hierarchy of behaviors, by collecting the various compositions of sub-behaviors, local channels and local variables. This commutative and associative relation is written as

$$r_1.r_2\dots r_n$$

where  $\forall i, 1 \leq i \leq n, r_i \in \bigcup\{R_c, R_{nw}, R_{nr}, R_t, R_{bw}, R_{br}, R_g\}$ .

Essentially, a grouping of model algebraic terms is another (more complex) term. Normal set theoretic operations like union and subtraction may be performed to manipulate complex terms formed by grouping. For example, we can use the dot (“.”) operation to add another term to a group as follows

$$(r_1.r_2\dots r_n).r_{n+1} = r_1.r_2\dots r_n.r_{n+1}$$

Similarly, we may subtract a term from a group as follows

$$(r_1.r_2\dots r_n) - r_i = r_1.r_2\dots r_{i-1}.r_{i+1}\dots r_n$$

Finally, we also define a *belongs-to* relation between sub-terms of a complex term and the complex term itself as follows

$$\forall i, \text{ such that } 1 \leq i \leq n, r_i \in r_1.r_2\dots r_n$$

## 3.2 Model Construction with MA

So far, we have seen the various objects and composition rules of MA. In this section, we look at how to construct hierarchical system models in MA using these objects and composition rules.

### 3.2.1 Hierarchy

Using the control flow relations, we can compose behaviors such that they execute in a desirable order. Most SDLs provide for hierarchical compositions of behaviors to aid modeling.

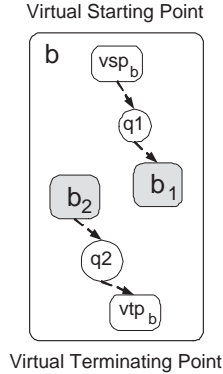


Figure 3.1: Control flow within hierarchical behaviors

In MA, hierarchy is achieved using the interface object and its relation to behaviors. In figure 3.1, a behavior  $b$  (shown as rounded rectangle) is created by hierarchical composition of sub-behaviors  $b_1$  and  $b_2$ . Note the *virtual starting point*(VSP) and the *virtual terminating point*(VTP) behaviors of  $b$ . The VSP is the identity behavior  $vsp_b$  that is the first to execute inside  $b$ . Other sub-behaviors of  $b$  are executed after  $vsp_b$ , depending on outgoing control relations (shown using broken arcs and circular nodes labeled with control conditions) from  $vsp_b$ . We can see in figure 3.1 that the VSP in this case  $vsp_b$  is triggering the execution of sub-behavior  $b_1$ . Due to its nature, a VSP behavior would only have outgoing control to other sub-behaviors of  $b$ . Likewise, the identity behavior  $vtp_b$  is the last behavior to execute inside  $b$ , and will only have incoming control from other sub-behaviors of  $b$ . All hierarchical behaviors are assumed to have a unique VSP and a VTP. Hence, the starting and terminating control relations of  $b$  can be written as

$$vsp_b \rightsquigarrow b_1.b_2 \rightsquigarrow vtp_b$$

### 3.2.2 Parallel and Conditional Execution

Most SLDLs provide for special constructs to create different types of behavioral hierarchies. The common ones are parallel composition and conditional composition. Figure 3.2(a) shows a parallel composition of behaviors  $b_1$  and  $b_2$ . A typical SLDL may allow construction of a parallel composition using a statement like

**par** {**run**  $b_1$ ; **run**  $b_2$ }.

Let the resulting behavior be called  $b_{par}$ . The execution of  $b_{par}$  indicates that both  $b_1$  and  $b_2$  are

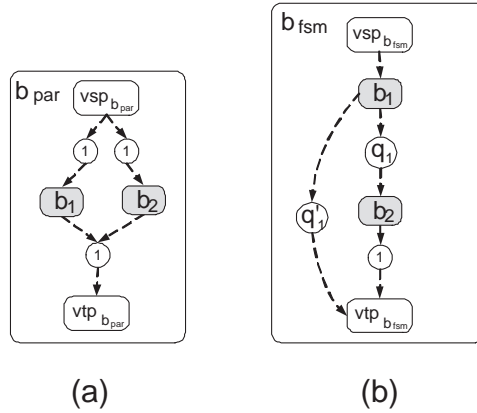


Figure 3.2: (a)Parallel and (b)FSM style compositions of behaviors

ready to execute. The execution of  $b_{par}$  completes when both  $b_1$  and  $b_2$  have completed. In the corresponding MA expression,  $vsp_{b_{par}}$  and  $vtp_{b_{par}}$  serve as the starting and terminating points, respectively, of the hierarchical behavior  $b_{par}$ . We can see, that inside  $b_{par}$ ,  $b_1$  and  $b_2$  are allowed to start simultaneously. This is ensured by the control relations

$$vsp_{b_{par}} \rightsquigarrow b_1.vsp_{b_{par}} \rightsquigarrow b_2$$

Hence, the parallelism is realized by orthogonality of the execution of behaviors  $b_1$  and  $b_2$ . The control relation at the end

$$b_1 \& b_2 \rightsquigarrow vtp_{b_{par}}$$

ensures that both  $b_1$  and  $b_2$  must complete their execution before  $vtp_{b_{par}}$  executes. The execution of  $vtp_{b_{par}}$  indicates the completion of the hierarchical behavior  $b_{par}$ .

A typical *if-then-else* style composition of behaviors is shown in Figure 3.2(b). A simple pseudo code example for a hierarchical behavior  $b_{fsm}$  is **run**  $b_1$ ; **if**  $q_1 == 1$  **goto** l2 **else** break; The control relations of  $b_{fsm}$  can be written as follows in MA

$$vsp_{b_{par}} \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow b_2.q_1' : b_1 \rightsquigarrow vtp_{b_{par}}.b_2 \rightsquigarrow vtp_{b_{par}}$$

### 3.2.3 Variable Access via Ports

In MA, a variable (shown using rectangular box) is directly visible only to the behaviors that are at the same scope of hierarchy as the variable itself. Therefore, in order to access variables

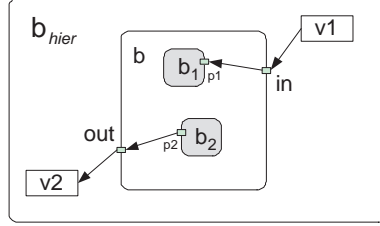


Figure 3.3: Using ports for non-blocking data flow in hierarchical behaviors

at higher levels of hierarchy, data ports are used. Ports are visualized as tiny boxes on the behavior interfaces. As shown in Figure 3.3, behavior  $b_1$  reads variable  $v_1$  present in  $b_{hier}$  via the port “in” of its parent  $b$ . Hence, to realized this port connection, we need terms at different levels of behavior hierarchy. At the scope of  $b_{hier}$ , we use the non-blocking relation  $v_1 \rightarrow b \langle in \rangle$ . At the scope of  $b$ , we use the port connection (shown using solid directed arcs) from the interface of  $b$  to  $b_1$ . We can write this as the relation

$$I \langle in \rangle \rightarrow b_1 \langle p_1 \rangle$$

The dual of read port connection is the write port connection as shown by the access of variable  $v_2$  from behavior  $b_2$  in figure 3.3. In this case, the port “out” of  $b$  is used to realize the variable access. The term at the scope of  $b_{hier}$  is

$$b \langle out \rangle \rightarrow v_2$$

while the term at the scope of  $b$  is

$$b_2 \langle p_2 \rangle \rightarrow I \langle out \rangle$$

### 3.2.4 Channel Access via Ports

As in the case of non-blocking reads and write, MA provides mechanism for blocking reads and writes (shown using solid arcs labeled with link address) via ports. For instance, in Figure 3.4, we see channel transactions from  $b$  to  $b'_1$  and  $b'_2$  over  $c$ , labeled  $a_1$  and  $a_2$  respectively. After zooming into the hierarchy of  $b$ , we see that the transactions are taking place from  $b_1$  to  $b'_1$  and  $b_2$  to  $b'_2$ . The ports  $p_1$  and  $p_2$  of  $b$  makes the channel  $c$  visible to  $b_1$  and  $b_2$ . Therefore, using the relation

$$\langle a \rangle : b_1 \langle p_{11} \rangle \mapsto I \langle p_1 \rangle$$

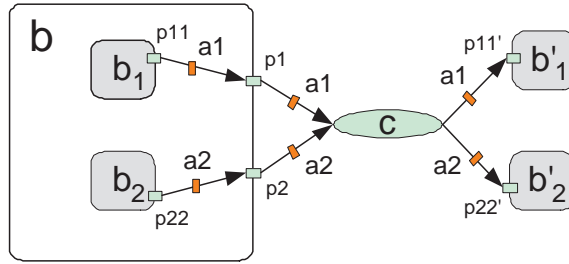


Figure 3.4: Sharing channel for transactions with different addresses

behavior  $b_1$  can access channel  $c$ . However, this requires  $p_1$  to be bound to the transaction link addressed by  $a_1$ .

In MA several virtual links may share a single channel. Each of the virtual links are assigned a different address, but the data transfer takes place on the same medium. Figure 3.4 shows an instance of channel sharing. Here, the two virtual links with addresses  $\langle a_1 \rangle$  and  $\langle a_2 \rangle$  use a common channel  $c$ . Transactions may be attempted concurrently on these links. However, due to sharing of the channel, we can allow only one transaction at a time. Thus, an arbiter in the channel must ensure that only one transaction may take place at any time. In MA, this is guaranteed by the mutual exclusion property of the channel, where the channel is a shared resource and each transaction is treated as a critical section. This allows us to connect several different virtual links to the same channel.

### 3.2.5 Using Identity Behaviors

A class of behaviors in MA is known as the identity behavior. As the name suggests, these behaviors have the same output as the input. As a result they do not have any computation inside them. They have two ports namely the “in” port for reading the input and an “out” port for writing the output. In general, the identity behavior first reads data from the “in” port to a local variable and then writes this variable to the “out” port. The actual implementation of the read and write within the identity behavior depends on the port connections.

There are four basic manifestations of the identity behavior as shown in figure 3.5. In the first case, as shown in figure 3.5(a), both the “in” and “out” ports of the identity behavior  $e_1$  are connected to variables. Hence, the respective read and write are non-blocking relations. In

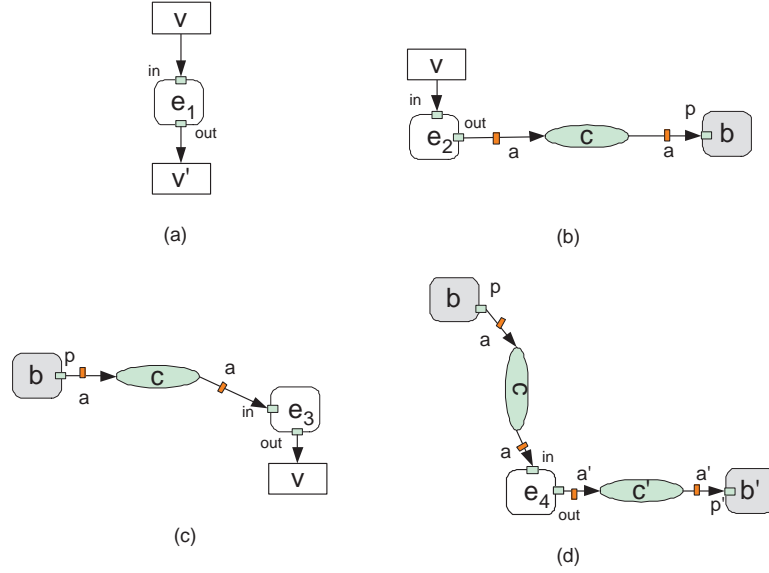


Figure 3.5: Various manifestations of the identity behavior

MA, the read/write relations of  $e_1$  are expressed as

$$v \rightarrow e_1 \langle in \rangle . e_1 \langle out \rangle \rightarrow v'$$

The second case of identity behavior is shown in figure 3.5(b). Here, the “in” port is connected to a variable, hence the input is read using a non-blocking relation. On the other hand, the “out” port is connected to channel  $c$ . Hence, the output needs to be sent to  $b$  using a blocking write relation. In MA, the read/write relations of  $e_2$  are expressed as

$$v \rightarrow e_2 \langle in \rangle . c \langle a \rangle : e_2 \langle out \rangle \mapsto b \langle p \rangle$$

The third case of identity behavior is shown in figure 3.5(c). Here, the “in” port is connected to a channel  $c$ , hence the input is read from behavior  $b$  using a channel transaction. On the other hand, the “out” port is connected to variable  $v$ . Hence, the output needs to be written using a non-blocking write relation. In MA, the read/write relations of  $e_3$  are expressed as

$$c \langle a \rangle : b \langle p \rangle \mapsto e_3 \langle in \rangle . e_3 \langle out \rangle \rightarrow v$$

Finally, the fourth manifestation of identity behavior is shown in figure 3.5(d). Here, the “in” port of  $e_4$  is connected to a channel  $c$  for reading data from  $b$ . Hence the input is read using

a channel transaction relation. The “out” port of  $e_4$  is also connected to a channel named  $c'$  for writing data to  $b'$ . Hence, the output is also written using a channel transaction relation. In MA, the read/write relations of  $e_4$  are expressed as

$$c \langle a \rangle : b \langle p \rangle \mapsto e_4 \langle in \rangle . c' \langle a' \rangle : e_4 \langle out \rangle \mapsto b' \langle p' \rangle$$

### 3.2.6 Hierarchical Modeling in MA

The model of a system is simply a behavior in MA. Typically, it is a hierarchical behavior showing the various components and connections of the system and the functionality within these components.

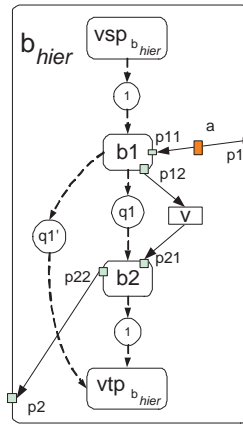


Figure 3.6: A hierarchical behavior with local objects and relations

#### 3.2.6.1 Internal and Interface Terms

In MA, it is possible to represent a hierarchical behavior as a grouping of terms involving its sub-behaviors, its interface and its local variables and channels. Figure 3.6 shows a hierarchical behavior  $b_{hier}$ . The expression for the hierarchical behavior is written using the local objects and their compositions. For instance, in the given behavior  $b_{hier}$ , we can see sub-behaviors  $b_1$  and  $b_2$ . We can also see control flow relations that determine the execution scenario under the conditions labeled on the control arcs. We also see data flow relations, both amongst sub-behaviors and between sub-behaviors and the interface. The grouping of relations between local objects will be referred to as the *internal terms* of a hierarchical behavior. Similarly, the grouping of relations

involving the interface will be referred to as the *interface terms* of the hierarchical behavior.

We can write the hierarchical behavior as a grouping of all its internal and interface terms, along with the internal terms of its sub-behaviors. The grouping of internal terms for a given behavior  $b$  is represented as  $[b]$ . Thus, we can write

$$[b_{hier}] = [vsp_{b_{hier}}].[b_1].[b_2].[vtp_{b_{hier}}].vsp_{b_{hier}} \rightsquigarrow b_1.q_1 : b_1 \rightsquigarrow b_2.$$

$$q'_1 : b_1 \rightsquigarrow vtp_{b_{hier}}.b_2 \rightsquigarrow vtp_{b_{hier}}.b_1 < p_{12} > \rightarrow v.v \rightarrow b_2 < p_{21} >$$

The interface terms of  $b_{hier}$  are represented by  $|b_{hier}|$ . From figure 3.6, we can see that

$$|b_{hier}| = \langle a \rangle : I \langle p_1 \rangle \mapsto b_1 \langle p_{11} \rangle . b_2 \langle p_{22} \rangle \rightarrow I \langle p_2 \rangle$$

Finally, we write the hierarchical behavior as a grouping of its internal and interface terms. We will use the convention of enclosing the expression for a hierarchical behavior in braces. Therefore, we get

$$b_{hier} = ([b_{hier}].|b_{hier}|)$$

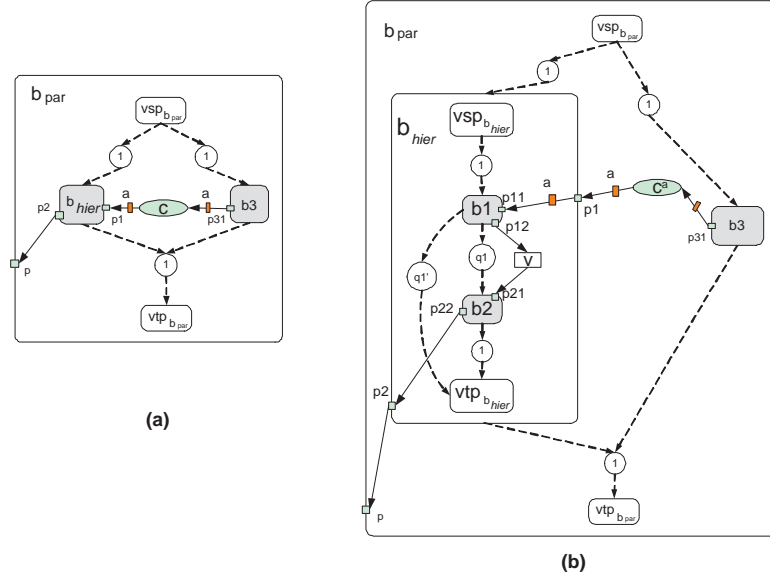


Figure 3.7: Hierarchical behavior  $b_{par}$  showing (a) local scope, and (b) all scopes down to leaf behaviors

### 3.2.6.2 Multiple Levels of Hierarchy

In the above example, a conditional hierarchical composition was created. The resulting behavior  $b_{hier}$  can be used further to create more hierarchical behaviors. For instance, in Figure 3.7(a), we see behavior  $b_{hier}$  in a parallel composition with behavior  $b_3$ . The two behaviors exchange data using the virtual link addressed  $a$ , over channel  $c$ . The hierarchical composition results in a new behavior called  $b_{par}$ . The expression for  $b_{par}$  is written in MA as follows

$$b_{par} = ([vsp_{b_{par}}].[b_{hier}].b_3.[vt_{pb_{par}}].vsp_{b_{par}} \rightsquigarrow b_{hier}.vsp_{b_{par}} \rightsquigarrow b_3.b_{hier} \& b_3 \rightsquigarrow vt_{pb_{par}}.$$

$$c < a > : b_3 < p_{31} > \mapsto b_{hier} < p_1 > . b_{hier} < p_2 > \rightarrow I < p >$$

The model of the system can be expressed in Model Algebra using a set of equations (as above) where each hierarchical behavior is expressed as a grouping of its local behaviors and control and data dependency terms.

## 3.3 Execution Semantics

In order to define the execution semantics of MA, the control relations in the model are captured using the *Behavior control graph*(BCG). BCG is simply a graph representation of the control flow in the model. Control dependencies implied by the rendezvous transactions on channels are also converted into BCG arcs to derive an execution model of the system. In this section we will first look at the formal execution semantics of model algebra using the BCG. Then we will look at semantics of channel transactions and how the control dependencies of such transaction links can be translated to BCG form.

### 3.3.1 Behavior Control Graph

The BCG is similar in principle to the Kahn Process Network [Kah74], but with some remarkable differences. It is a directed graph  $BCG(N,E)$  with two types of nodes, namely *behavior nodes*( $N_B$ ) and *control nodes*( $N_Q$ ). The behavior nodes, as the name suggests, indicate behavior execution, while the control nodes represent evaluation of control conditions that lead to further behavior executions. Directed edges are allowed from behavior nodes to control nodes and vice

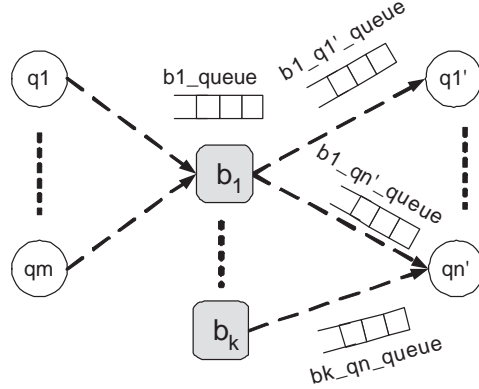


Figure 3.8: The firing semantics of BCG nodes

versa. Also, a control node can have one, and only one, out going edge. Thus,

$$E(BCG) \subset N_B(BCG) \times N_Q(BCG) \cup N_Q(BCG) \times N_B(BCG)$$

There is a one to one correlation between the BCG and the control relations of the MA representation. A relation of the form

$$q : b_1 \& b_2 \& \dots \& b_n \rightsquigarrow b$$

translates to a BCG with a control node ( $q$ ),  $n + 1$  behavior nodes ( $b, b_1, \dots, b_n$ ) and  $n + 1$  directed arcs  $((b_1, q), \dots, (b_n, q), (q, b))$ .

The execution of a behavior node, and similarly, the evaluation in a control node, will be referred to as a *firing*. Node firings are facilitated by tokens that circulate in the queues of the BCG as shown in Figure 3.8. Each behavior node (shown by rounded edged box) in the BCG has one queue, for instance  $b1\_queue$  for behavior node  $b_1$ . All incoming edges to a behavior node represent the various writers to the behavior's queue. The control node (shown by circular node), on the other hand, has as many queues as the number of incoming edges. Note that a control node may have only one outgoing edge. This restriction follows directly from the control flow composition rule of model algebra. As shown in Figure 3.8  $q_n$  has  $k$  queues, one each for edges from  $b_1$  through  $b_k$ .

A behavior node blocks on an empty queue and fires if there is at least one token in its queue. Upon firing, one token is dequeued from the node's queue. After firing, a behavior

node first updates all variables that it is writing to. Then, it generates as many tokens as its out-degree, and each token is written to the corresponding queue of the destination control node in a non-blocking fashion.

A control node, sequentially checks all its queues and blocks on empty queues. That is if an incoming edge's queue is empty, we must wait until the behavior on the other end of the edge has fired. If the queue is not empty, it dequeues a token from the queue and proceeds to check the next queue. Once we have dequeued one token from each of the incoming queues, the condition inside the control node is evaluated. If the result is TRUE, then the control node fires. Upon firing, the control node enqueues one token to the queue of the destination behavior in a non-blocking fashion. The process of checking the incoming queues is repeated indefinitely.

### 3.3.2 Channel Semantics

The channel object allows for rendezvous communication [Hoa85, Mil80] between two concurrently executing behaviors. As discussed before, a channel transaction implies a control dependency between parts of the communicating behaviors. Without loss of generality, we can assume both the sender and the receiver to be identity behaviors. We will follow this assumption in all future discussions.

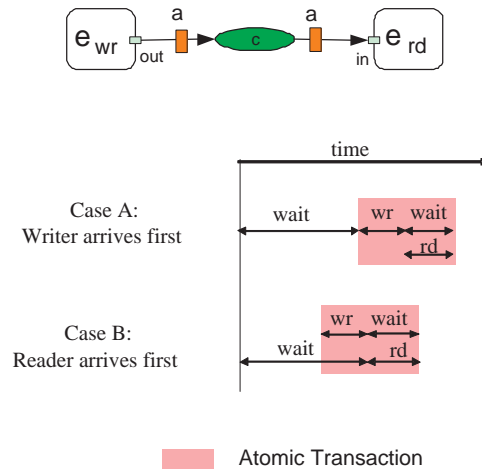


Figure 3.9: Timing diagram of a transaction on a channel

### 3.3.2.1 Channels with single transaction link

Figure 3.9 shows a transaction taking place over channel  $c$ . We can express this transaction MA using the term  $c \langle a \rangle: e_{wr} \langle out \rangle \mapsto e_{rd} \langle in \rangle$ . The timing diagram for this channel transaction shows two instances of execution. In the first instance, called Case A, the writer reaches the communication point before the reader. By this we mean that during model execution,  $e_{wr}$  is scheduled to execute before  $e_{rd}$ . However, the rendezvous semantics dictate that  $e_{wr}$  must wait until  $e_{rd}$  is ready before executing. It may be noted that if there is a control dependency from  $e_{wr}$  to  $e_{rd}$ , the resulting model would deadlock. Hence,  $e_{rd}$  must be allowed to start independently of  $e_{wr}$  and vice versa. Once  $e_{rd}$  is ready to start the transaction, it notifies  $e_{wr}$ . The transaction is thus initiated by  $e_{wr}$ , that performs a write on the channel. Subsequently,  $e_{rd}$  reads the data from the channel.

In the second execution scenario, called Case B, the reader is scheduled before the writer is ready. This forces  $e_{rd}$  to wait until  $e_{wr}$  is ready to start executing. The shaded part of the execution, in the timing diagram, indicates the atomic nature of the transaction. Note that the channel resources (i.e. its local memory) are occupied only during the actual reading and writing of the data, not during synchronization.

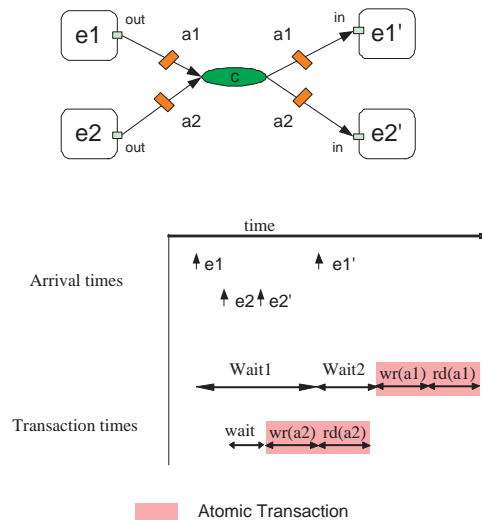


Figure 3.10: Multiple competing transactions on a single channel

### 3.3.2.2 Channels with multiple transaction links

As discussed earlier, channel sharing is possible for different virtual links, but the transactions are ordered in time. This mutual exclusivity of transactions can be achieved by the use of semaphores or arbiters. Thus, the shaded part representing the actual data read and write over the local memory of channel is mutually exclusive. Consider the configuration shown in figure 3.10. In this case, two virtual links, addressed  $a_1$  and  $a_2$ , are shared over channel  $c$ . These links can be written as a grouping of the following terms

$$c \langle a_1 \rangle : e_1 \langle out \rangle \mapsto e'_1 \langle in \rangle . c \langle a_2 \rangle : e_2 \langle out \rangle \mapsto e'_2 \langle in \rangle$$

The timing diagram shows the actual arrival schedule of the four communicating identity behaviors and the resulting communication schedule on the channel. Note that despite the fact that  $e_1$  arrives first, transaction on  $a_2$  takes place before that on  $a_1$ . This is because, the data transfer of transaction addressed  $a_2$  is ready to be performed before that for  $a_1$ . Thus, the data transfers on the channel are scheduled on first-ready first-serve basis. Although the transaction on  $a_1$  is ready to be performed when  $e'_1$  arrives, it must wait for the duration  $wait_2$  since the transaction addressed  $a_2$  is in progress.

### 3.3.3 Control paths and dominators

We will now define some useful control flow relations between behaviors in a model, based on the model's BCG. A control path from a behavior  $b_1$  to  $b_2$  in model  $M$  is simply a non-zero length path in the BCG of  $M$  from node representing  $b_1$  to node representing  $b_2$ . If such a path exists, then  $b_2$  is said to be *reachable* from  $b_1$ , else  $b_2$  is said to be *unreachable* from  $b_1$ .

A behavior  $b$  is said to *dominate* control flow from  $b_1$  to  $b_2$  in a given model  $M$ , if during execution of  $M$ ,  $b$  always fires at least once *between* any two unique firings of  $b_1$  and  $b_2$ . This relation between  $b$ ,  $b_1$  and  $b_2$  is written as

$$b \triangleright (b_1, b_2)$$

It must be noted that the dominator relation does not mean that  $b$  must exist in all control paths from  $b_1$  to  $b_2$ . Figure 3.11 shows BCG of a simple model with four behaviors  $b_1$  through  $b_4$ . The control paths from  $b_1$  to  $b_4$  are indicated by thick broken arrows labeled CP1 and CP2. It can be easily seen that  $b_2 \triangleright (b_1, b_4)$ , since any firing of  $b_4$  is always preceded by a firing of  $q_4$  and  $q_4$  does

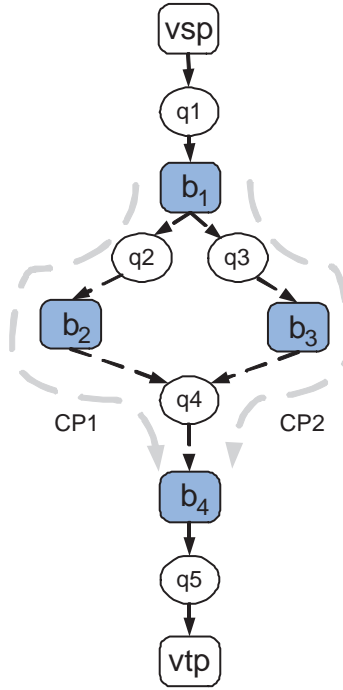


Figure 3.11: Illustration of control paths

not fire until  $b_2$  has fired. However,  $b_2$  does not exist in the control path  $(b_1, q_3, b_3, q_4, b_4)$  (labeled CP2).

A special case of the dominator relation is as follows. Assume there exist behaviors  $b$  and  $b'$  in model  $M$  such that during any execution of  $M$ , there is atleast one unique firing of  $b$  **before** every unique firing of  $b'$ . We say that  $b$  dominates  $b'$  and we will write the control relation between  $b$  and  $b'$  as  $b \triangleright b'$ . It can be easily seen that

$$b \triangleright b' \Leftrightarrow (b \triangleright (vsp, b') \wedge b \triangleright (b', b'))$$

The RHS of the above equation indicated that  $b$  must fires once between the firing of  $vsp$  (start of model execution) and the (first) firing of  $b'$ . Any subsequent firings of  $b'$  (after its first firing) should also follow a unique firing of  $b$ , as guaranteed by relation  $b \triangleright (b', b')$ . This is equivalent to  $b \triangleright b'$  by definition of the dominator relation.

From the above definitions, we can also make the following observations

$$b_1 \triangleright b_2 \wedge b_2 \triangleright b_3 \Rightarrow b_2 \triangleright (b_1, b_3) \tag{3.1}$$

$$b_1 \triangleright b_2 \wedge b_2 \triangleright b_3 \Rightarrow b_1 \triangleright b_3 \tag{3.2}$$

## 3.4 Notion of Functional Equivalence

Before we begin to verify models written in MA, we must first establish a formal notion of equivalence. There are several possible notions of equivalence for process algebras like trace equivalence [BNP02], bisimulation equivalence [Mil80] and so on. The basic purpose of a formal equivalence notion is to be able to distinguish different models written in a formalism. An equivalence notion is said to be *stronger* if it distinguishes more models. For instance two models may be *trace equivalent*, but not *bisimulation equivalent*. However, all models that are bisimulation equivalent are also trace equivalent. Therefore, bisimulation is a *stronger* notion of equivalence compared to trace equivalence. Conversely, trace equivalence is *weaker* than bisimulation. The strongest possible equivalence is *isomorphism* which indicates a syntactic equality of two models. An interesting comparison of formal notions of equivalence may be found in [vGG89]

### 3.4.1 Value Traces

Our notion of functional equivalence is based on the trace of values that certain interesting variables hold during model execution. We will refer to such variables as *observed variables*. Given a model  $M$  and a set of its observed variables, say  $OV_M$ , let  $Init_M$  be the initial assignment of all the variables in  $OV_M$ . Let  $v \in OV_M$ . We define  $\tau_M(v, Init_M)$  as the set of all possible traces of values assumed by  $v$ , when model  $M$  is executed with initialization  $Init_M$ .

Figure 3.12 shows a simple model, say  $M$ . We have shown the function inside leaf level behaviors to demonstrate the trace values. Let  $x$  and  $y$  be the observed variables, with an initial assignment of 0 and  $U$  (undefined). When model  $M$  is executed,  $b_1$  fires after  $vsp$ , setting the value of  $y$  to  $2 * 0 = 0$ . since  $x < 2$ ,  $b_2$  fires and increments  $x$  to 1. This is followed by another firing of  $b_1$  and so on. Hence, for the given scenario, we have

$$\begin{aligned} OV_m &= \{x, y\} \\ Init_M &= \{0, U\} \\ \tau_M(x, Init_M) &= \{\{0, 1, 2, 3\}\} \\ \tau_M(y, Init_M) &= \{\{U, 0, 2, 4, 6\}\} \end{aligned}$$

The model terminates with  $x = 3$  and  $y = 6$ . However, it is possible that model execution

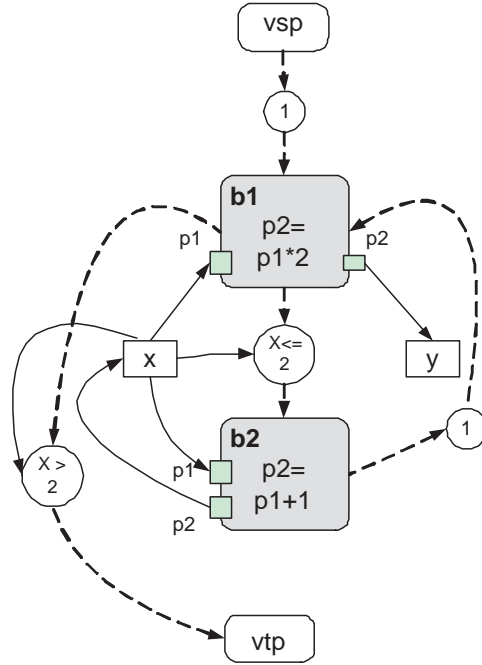


Figure 3.12: Example of a model ( $M$ ) with unique value trace

may never terminate and the traces are infinitely long.

Figure 3.13 shows a new model, say  $M'$ . In this model, the behaviors  $b_1$  and  $b_2$  are rearranged to execute in parallel. As in the previous case, let  $x$  and  $y$  be the observed variables, with an initial assignment of 0 and  $U$  (undefined). When model  $M'$  is executed, either  $b_1$  or  $b_2$  may fire first after  $e$ . Therefore, the value of  $y$  may become 0 if  $b_1$  fires first, or become 2 if  $b_2$  fires first (modifying  $x$  to 1). However, by the time  $e_2$  executes  $x$  is definitely 1. Since  $x < 2$ ,  $e_1$  fires again, allowing either  $b_1$  or  $b_2$  to fire first. The process continues until  $x$  becomes 3, Hence, for the given scenario, we have

$$\begin{aligned}
 OV_m &= \{x, y\} \\
 Init_M &= \{0, U\} \\
 \tau_M(x, Init_M) &= \{\{0, 1, 2, 3\}\} \\
 \tau_M(y, Init_M) &= \{\{U, 0, 2, 4\}, \{\{U, 0, 2, 6\}, \{\{U, 0, 4\}, \\
 &\quad \{\{U, 0, 4, 6\}, \{\{U, 2, 4\}, \{\{U, 2, 6\}, \{\{U, 2, 4, 6\}\}\}
 \end{aligned}$$

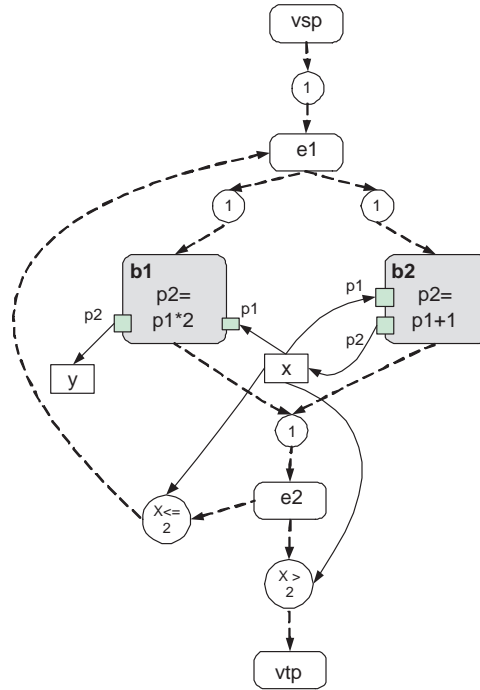


Figure 3.13: Example of a model ( $M'$ ) with several possible value traces

### 3.4.2 Trace based equivalence

Given two models  $M$  and  $M'$  expressed using model algebra, we wish to determine if they are equivalent with respect to some well defined notion of equivalence. First, we need to determine a correlation between the two models based on their respective observed variables. We choose the observed variables of a model as the set of all variables in a model that are not written exclusively by identity behaviors. The reasoning is that variables that are connected to the output ports of identity behaviors are simply a copy of another variable. Informally speaking, we consider two models to be functionally equivalent, if they have one-to-one correspondence of observed variables and the trace of values assumed by those variables during model execution is identical, given the same initial assignment. The traces should be identical for all possible initial assignments.

More formally, in order to show equivalence of  $M$  and  $M'$ , we require that,

$$|OV_M| = |OV_{M'}|$$

and there exists a bijective mapping from  $OV_M$  to  $OV_{M'}$ . We will represent the mapping of respec-

tive elements by the  $\leftrightarrow$  symbol. Therefore, we have

$$\forall v \in OV_M, \exists v' \in OV_{M'}, \text{ such that } v \leftrightarrow v'$$

Let  $Init_M$  be the initial assignment of all variables in  $OV_M$  and  $Init_{M'}$  be the initial assignment of all variables in  $OV_{M'}$ . Models  $M$  and  $M'$  are equivalent if

$$\forall Init_M, Init_{M'}, \forall v \in OV_M, v' \in OV_{M'}, \text{ such that } v \leftrightarrow v'$$

if the initial assignment of  $v$  is equal to the initial assignment of  $v'$ , then

$$\tau_M(v, Init_M) = \tau_{M'}(v', Init_{M'})$$

### 3.5 Transformation laws of MA

We will now define laws of MA that will allow us to perform useful functionality preserving transformations on a model. For clarity, We will demonstrate these transformations on the graphical representations of the model. The transformations can also be shown on corresponding MA expressions, since the two representations have one-to-one correlation.

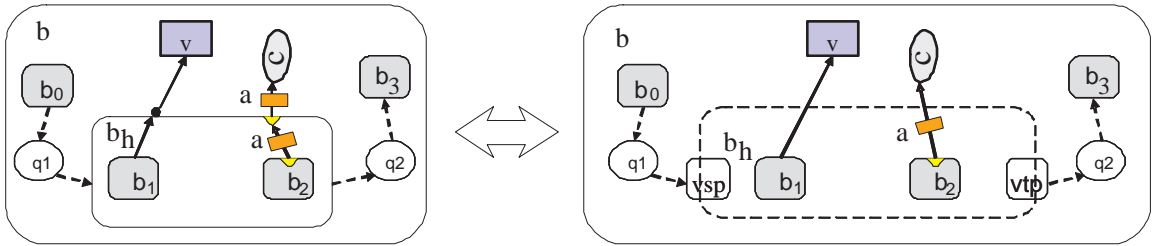


Figure 3.14: Illustration of flattening law

#### 3.5.1 Flattening of Hierarchical Behaviors

Addition of hierarchy allows the designer to group different behaviors together. It does not add any functionality to the model. For functional validation, we need to be concerned with only the leaf level behaviors. Hence, we may get rid of hierarchy by flattening the model. The law

for flattening a hierarchical behavior follow from the semantics of hierarchical behaviors in MA. Consider the hierarchical behavior  $b_h$  in figure 3.14. According to the semantics of the VSP, any control relation leading to  $b_h$  is effectively leading to  $vsp$ , where  $vsp$  is the virtual starting point of behavior  $b_h$ . This is because  $vsp$  is always the first behavior to execute inside  $b_h$ . Similarly, in any control relation where  $b_h$  is a predecessor, it may be replaced by  $vtp$ . This is because  $vtp$  is always the last behavior to execute inside  $b_h$ .

To enable data flow, hierarchical behaviors allow for ports on their interface. These ports are essentially a conduit for data transfer from one leaf behavior to another. During flattening, these ports can be optimized away by appropriately making new port connections as shown in RHS of Figure 3.14. A virtual link addressed  $a$  over channel  $c$  is used for blocking data transfer from  $b_2$ . However, due to the hierarchical behavior  $b_h$ , channel  $c$  is not visible from the local scope of  $b_2$ . Thus, a port is used to facilitate the connection of  $b_2$  with channel  $c$ . When the interface of  $b_h$  is removed during flattening, we can directly connect  $b_2$  to channel. Similarly,  $b_1$  may be directly connecting to variable  $v$  during flattening. Boolean functions for control conditions inside the hierarchical behavior may have in-ports as parameters. During flattening, when in-ports are replaced by their port maps, the boolean functions for the control conditions must be updated by replacing the in-port parameter with the port map.

### 3.5.1.1 Flattening rule

The flattening transformation rule consists of several syntactic manipulations of model algebraic expression as shown below. Assume that  $b_h$  is the hierarchical behavior which is a sub-behavior of  $b$  that we are trying to flatten. Let  $vsp_h$  be the virtual starting point of  $b_h$  and  $vtp_h$  be the virtual terminating point of  $b_h$ . It must be noted that the flattening transformation is atomic, that means all the following syntactic manipulations must take place together in a single transformation step.

- (i)  $\forall q, b_1, \dots, b_n$ , such that  $q : b_1 \& \dots \& b_n \rightsquigarrow b_h \in b$ ,  
 $b = (b - q : b_1 \& \dots \& b_n \rightsquigarrow b_h). q : b_1 \& \dots \& b_n \rightsquigarrow vsp_h$
- (ii)  $\forall q, b_1, \dots, b_n, b'$  such that  $q : b_1 \& \dots \& b_h \& \dots \& b_n \rightsquigarrow b' \in b$ ,  
 $b = (b - q : b_1 \& \dots \& b_h \& \dots \& b_n \rightsquigarrow b'). q : b_1 \& \dots \& vtp_h \& \dots \& b_n \rightsquigarrow b'$

- (iii)  $\forall p, p_h, p' \in \mathcal{P}$ , such that  $b_h \langle p_h \rangle \rightarrow I \langle p \rangle \in b \wedge b' \langle p' \rangle \rightarrow I \langle p_h \rangle \in b_h$ ,  
 $b = b.(b' \langle p' \rangle \rightarrow I \langle p \rangle)$ ,  $b = b - (b_h \langle p_h \rangle \rightarrow I \langle p \rangle)$
- (iv)  $\forall p', p_h \in \mathcal{P}, v \in \mathcal{V}$ , such that  $b_h \langle p_h \rangle \rightarrow v \in b \wedge b' \langle p' \rangle \rightarrow I \langle p_h \rangle \in b_h$ ,  
 $b = b.(b' \langle p' \rangle \rightarrow v)$ ,  $b = b - (b_h \langle p_h \rangle \rightarrow v)$
- (v)  $\forall p, p_h, p' \in \mathcal{P}, q \in \mathcal{Q}$  such that  $I \langle p \rangle \rightarrow b_h \langle p_h \rangle \in b \wedge I \langle p_h \rangle \rightarrow b' \langle p' \rangle \in b_h, q = f_b(\dots I \langle p_h \rangle \dots)$ ,  
 $b = b.(I \langle p \rangle \rightarrow b' \langle p' \rangle)$ ,  $b = b - (I \langle p \rangle \rightarrow b_h \langle p_h \rangle)$ ,  $q = f_b(\dots I \langle p \rangle \dots)$
- (vi)  $\forall p', p_h \in \mathcal{P}, v \in \mathcal{V}, q \in \mathcal{Q}$  such that  $v \rightarrow b_h \langle p_h \rangle \in b \wedge I \langle p_h \rangle \rightarrow b' \langle p' \rangle \in b_h, q = f_b(\dots I \langle p_h \rangle \dots)$ ,  
 $b = b.(v \rightarrow b' \langle p' \rangle)$ ,  $b = b - (v \rightarrow b_h \langle p_h \rangle)$ ,  $q = f_b(\dots I \langle p \rangle \dots)$
- (vii)  $\forall a \in \mathcal{A}, p, p_h, p' \in \mathcal{P}$ , such that  $a : b_h \langle p_h \rangle \mapsto I \langle p \rangle \in b \wedge a : b' \langle p' \rangle \mapsto I \langle p_h \rangle \in b_h$ ,  
 $b = b.(a : b' \langle p' \rangle \mapsto I \langle p \rangle)$ ,  $b = b - (a : b_h \langle p_h \rangle \mapsto I \langle p \rangle)$
- (viii)  $\forall a \in \mathcal{A}, p_{h'}, p', p_h \in \mathcal{P}, c \in \mathcal{C}$ , such that  $c \langle a \rangle : b_h \langle p_h \rangle \rightarrow b_{h'} \langle p_{h'} \rangle \in b$   
 $\wedge a : b' \langle p' \rangle \mapsto I \langle p_h \rangle \in b_h$ ,  
 $b = b.(c \langle a \rangle : b' \langle p' \rangle \mapsto b_{h'} \langle p_{h'} \rangle)$ ,  $b = b - (c \langle a \rangle : b_h \langle p_h \rangle \rightarrow b_{h'} \langle p_{h'} \rangle)$
- (ix)  $\forall a \in \mathcal{A}, p, p_h, p' \in \mathcal{P}$ , such that  $a : I \langle p \rangle \mapsto b_h \langle p_h \rangle \in b \wedge a : I \langle p_h \rangle \mapsto b' \langle p' \rangle \in b_h$ ,  
 $b = b.(a : I \langle p \rangle \mapsto b' \langle p' \rangle)$ ,  $b = b - (a : I \langle p \rangle \mapsto b_h \langle p_h \rangle)$
- (x)  $\forall a \in \mathcal{A}, p_{h'}, p', p_h \in \mathcal{P}, c \in \mathcal{C}$ , such that  $c \langle a \rangle : b_{h'} \langle p_{h'} \rangle \mapsto b_h \langle p_h \rangle \in b$   
 $\wedge a : I \langle p_h \rangle \mapsto b' \langle p' \rangle \in b_h$ ,  
 $b = b.(c \langle a \rangle : b_{h'} \langle p_{h'} \rangle \mapsto b' \langle p' \rangle)$ ,  $b = b - (c \langle a \rangle : b_{h'} \langle p_{h'} \rangle \mapsto b_h \langle p_h \rangle)$

### 3.5.1.2 Soundness proof

The soundness of flattening rule follows directly from the definition of hierarchy in MA. By definition of the VSP of behavior  $b_h$ , firing of  $b_h$  starts with the firing of  $vsp_h$ . That is a token enqueued for  $b_h$  in the BCG of hierarchical model corresponds to a token enqueued for  $vsp_h$  in the BCG of flattened model. Similarly, the firing of  $b_h$  completes with the firing of its  $vt_p_h$ . Therefore,

we can say that

$$\forall q \in Q, x_i \in \mathcal{B}, q : x_1 \& x_2 \dots \& x_n \rightsquigarrow b_h = q : x_1 \& x_2 \dots \& x_n \rightsquigarrow vsp_h, \text{ and}$$

$$\forall q \in Q, x \in \mathcal{B}, q : \dots \& b_h \& \dots \rightsquigarrow x = q : \dots \& b_h \& \dots \rightsquigarrow vtp_h$$

Therefore, manipulations (i) and (ii) of the flattening rule are sound.

The firing of a behavior results in reading of variables that are mapped to its inport and the updation of variables mapped to its output. For all  $b'$  such that  $b'$  is a sub-behavior of  $b_h$ , the firing of  $b'$  updates all variables mapped to its output. If the out ports of  $b'$  are mapped to out ports of  $b_h$ , then the variables mapped to the latter will be updated. Therefore, by inductive reasoning, manipulations (iii) through (vi) are sound. Using a similar reasoning for port mappings for ports connected to channel, we establish the soundness of manipulations (vii) through (x).

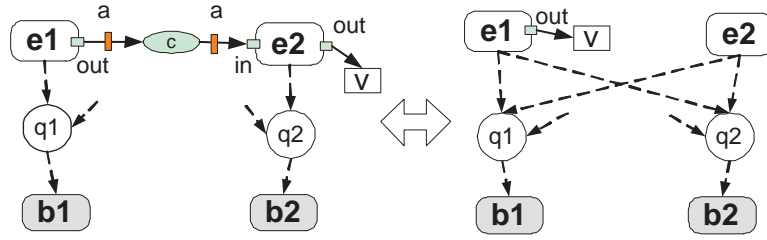


Figure 3.15: Resolution of channels into control dependencies

### 3.5.2 Control flow resolution of links

As seen during the discussion of channel semantics, the channels in MA imply control flow dependencies between communicating behaviors. Our eventual goal is to collect all control dependencies resulting from transaction links and incorporate them into the BCG. We will now see how to resolve the transaction links in MA models into control dependencies. Figure 3.15 demonstrates this control dependency extraction and the modification of data dependencies.

Recall our assumption that blocking relations and channel transaction relations can involve only identity behaviors or hierarchical behaviors. Thus, for the purpose of control flow extraction from channel transaction relations, we need to consider only the case where sender and receiver are both identity behaviors. The synchronization properties of a channel would ensure the following two premises:

1. Any behavior following the sender identity behavior would not execute until the receiver identity behavior has executed.
2. Any behavior following the receiver identity behavior would not execute until the sender identity behavior has executed.

If we were to optimize away the channel to extract only the control dependencies, the result will be as shown in figure 3.15. As per the above premises, behavior  $b_1$  following sender  $e_1$  cannot start until  $e_2$  has completed. This is guaranteed by adding the arc  $(e_2, q_1)$  to the BCG. In the dual of the above case,  $b_2$  following  $e_2$  is blocked until the sender  $e_1$  has executed. This premise is ensured by adding the arc  $(e_1, q_2)$ .

### 3.5.2.1 Link resolution rule

Assume that  $b$  is a hierarchical behavior, with identity sub-behaviors  $e_1$  and  $e_2$ . We will also assume that the input and output ports of both these identity behaviors are named *in* and *out* respectively. Let there be a channel transaction link from  $e_1$  to  $e_2$  over channel  $c$ , with address  $a$ . Also, let  $e_2$  write to variable  $v$ . Hence, we have

$$c < a > : e_1 < out > \mapsto e_2 < in > \in b \wedge e_2 < out > \rightarrow v \in b$$

The link resolution rule consists of several syntactic manipulations of model algebraic expression as shown below. It must be noted that the flattening transformation is atomic, that means all the following syntactic manipulations must take place together in a single transformation step.

- (i)  $\forall q \in Q, b_1 \in \mathcal{B}$ , such that  $q : \dots \& e_1 \& \dots \rightsquigarrow b_1 \in b$ ,  
 $b = (b - q : \dots \& e_1 \& \dots \rightsquigarrow b_1). q : \dots \& e_1 \& e_2 \& \dots \rightsquigarrow b_1$
- (ii)  $\forall q \in Q, b_2 \in \mathcal{B}$ , such that  $q : \dots \& e_2 \& \dots \rightsquigarrow b_2 \in b$ ,  
 $b = (b - q : \dots \& e_2 \& \dots \rightsquigarrow b_2). q : \dots \& e_1 \& e_2 \& \dots \rightsquigarrow b_2$
- (iii)  $b = b - c < a > : e_1 < out > \mapsto e_2 < in >$
- (iv)  $b = (b - e_2 < out > \rightarrow v). e_1 < out > \rightarrow v$

### 3.5.2.2 Soundness proof

The soundness of link resolution rule follows from the definition of transaction semantics for channels. Assume a model  $M$  with transaction

$$c \langle a \rangle : e_1 \langle out \rangle \mapsto e_2 \langle in \rangle$$

From the channel semantics, firing of  $e_1$  is not complete until firing of  $e_2$  is complete and vice versa. In other words, all control nodes in the BCG of  $M$  that have edges from  $e_1$  will not fire until  $e_2$ . Note however, that any such control node, say  $q$ , may not have an explicit edges from  $e_2$ . The synchronization semantics of the transaction link have the same effect on firing of  $q$ , as if there were a an extra edge  $(e_2, q)$  in the BCG of  $M$ . Therefore, the LHS and RHS in manipulations (i) and (ii) have the same firing results in the respective BCGs. Hence manipulations (i) and (ii) are sound.

By definition of identity behaviors in Section 3.2.5, the identity copies any incoming data (non-blocking or blocking read) to its out port in either a blocking or non-blocking way. Therefore, when  $e_1$  fires, it reads its in port and sends data to  $e_2$  via  $c$  and using address  $a$ . Assume that  $e_2$  writes to variable  $v$ . The channel transaction and relation  $e_2 \langle out \rangle \rightarrow v$  is equivalent to  $e_1 \rightarrow v$ , since  $e_1$  does not complete firing until  $e_2$ . Hence, manipulations (iii) and (iv) are sound.

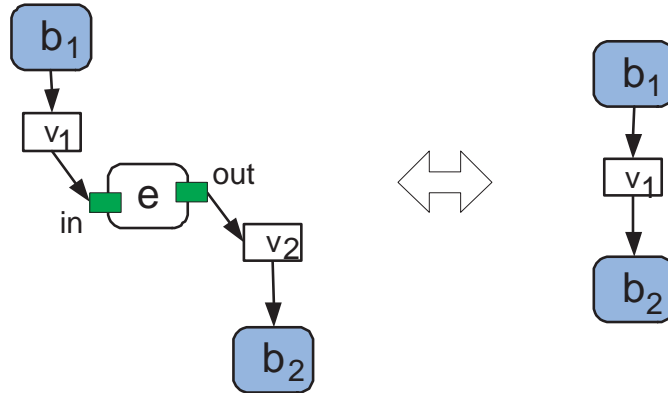


Figure 3.16: Variable merging

### 3.5.3 Variable merging

By definition, identity behaviors copy a value from one variable to another when they are fired. Data read relation from the copy variable may be replaced by data read relation from source variable using the variable merging rule. The source variable, say  $v_1$  is identified as the one which is connected to the in-port of the identity behavior, say  $e$  via the relation

$$v_1 \rightarrow e \langle in \rangle$$

where  $in$  is the inport of  $e$ . The copy behavior  $v_2$  is identified by its connection to the out-port of  $e$  by the relation

$$e \langle out \rangle \rightarrow v_2$$

Figure 3.16 illustrates the variable merging rule. On the LHS, the source variable  $v_1$  is written by behavior  $b_1$  while the copy behavior  $v_2$  is read by behavior  $b_2$ . After  $v_2$  is merged into  $v_1$ , behavior  $b_2$  can read the merged behavior  $v_1$  as shown on the RHS. Such merging of variables is possible only if the identity behavior  $e$  fires at least once between any firing of  $b_1$  and  $b_2$ . In other words,  $e$  must dominate the control flow from  $b_1$  to  $b_2$ .

The merging of variables also has implications on the control conditions. The control condition is nothing but a boolean function of the variables in the behavior. If a control condition, say  $q$ , depends on the copy variable, then during merging the copy variable must be replaced by the merged variable in the boolean function of  $q$ .

#### 3.5.3.1 Variable merging rule

Assume that  $b$  is a hierarchical behavior, with identity sub-behavior  $e$  and variables  $v_1$  and  $v_2$  such that

$$v_1 \rightarrow e \langle in \rangle \in b \wedge e \langle out \rangle \rightarrow v_2 \in b \wedge \nexists b' (\neq e) \in \mathcal{B}, p' \in \mathcal{P}, \text{ s.t. } b' \langle p' \rangle \rightarrow v_2 \in b$$

Also  $\forall b_1, b_2 \in \mathcal{B}, p_1, p_2 \in \mathcal{P}$ , such that

$$b_1 \langle p_1 \rangle \rightarrow v_1 \wedge v_2 \rightarrow b_2 \langle p_2 \rangle$$

if  $e \triangleright (b_1, b_2)$ , then the following syntactic manipulations may be performed for variable merging

$$(i) \forall b_2 \in \mathcal{B}, p_2 \in \mathcal{P}, b = (b - v_2 \rightarrow b_2 < p_2 >). v_1 \rightarrow b_2 < p_2 >$$

$$(ii) \forall q = f_b(\dots, v_2, \dots), q = f_b(\dots, v_1, \dots)$$

$$(iii) b = (b - e < out > \rightarrow v_2) - v_1 \rightarrow e < in >$$

### 3.5.3.2 Soundness proof

The soundness of variable merging can be proved as follows. From the assumptions for application of the rule, we know that  $e$  is the only writer to  $v_2$ . Also, by the dominator relation,  $e$  fires at least once between a firing of any writer of  $v_1$  and firing of any reader of  $v_2$ . Let  $b_1$  be a writer of  $v_1$  and  $b_2$  be a reader of  $v_2$ . When  $b_1$  fires, it updates  $v_1$  with a value, say  $l$ . After  $b_1$  has fired, and before any firing of  $b_2$ , identity  $e$  fires and copies  $l$  into  $v_2$ . Therefore, for  $b_2$ , it does not make a difference if it is reading  $v_1$  or  $v_2$ , since it will read the same absolute value. We will assume without loss of generality that no other writer of  $v_1$  and no other reader of  $v_2$  fire meanwhile.

We also know that  $v_2$  is not an observed variable in the model, since it does not have a non-identity writer. Since the control dependencies do not change during variable merging, the firing partial orders of the behaviors will also not change. By our previous argument, under the given assumptions, all the behaviors continue to read the same values as before for all their firings. Therefore, the firing results guarantee the same trace of all observed variables before and after variable merging. Hence, the variable merging transformation rule is sound.

### 3.5.4 Identity elimination

The identity behavior, by definition, does not perform any computation. If an identity behavior does not write any variables in the model, it may be optimized away using the identity elimination rule.

The example illustrated in figure 3.17 shows three different cases in which identity elimination can be applied. In Case (a), on the LHS we have an identity behavior  $e$ , with 1 incoming control edge from control node  $q_1$  and  $m$  outgoing control edges to control nodes  $q'_1$  through  $q'_m$ . As illustrated in the figure, the outgoing control nodes may have control edges to it from behaviors other than  $e$ . After optimization of  $e$  using identity elimination, the incoming control node is

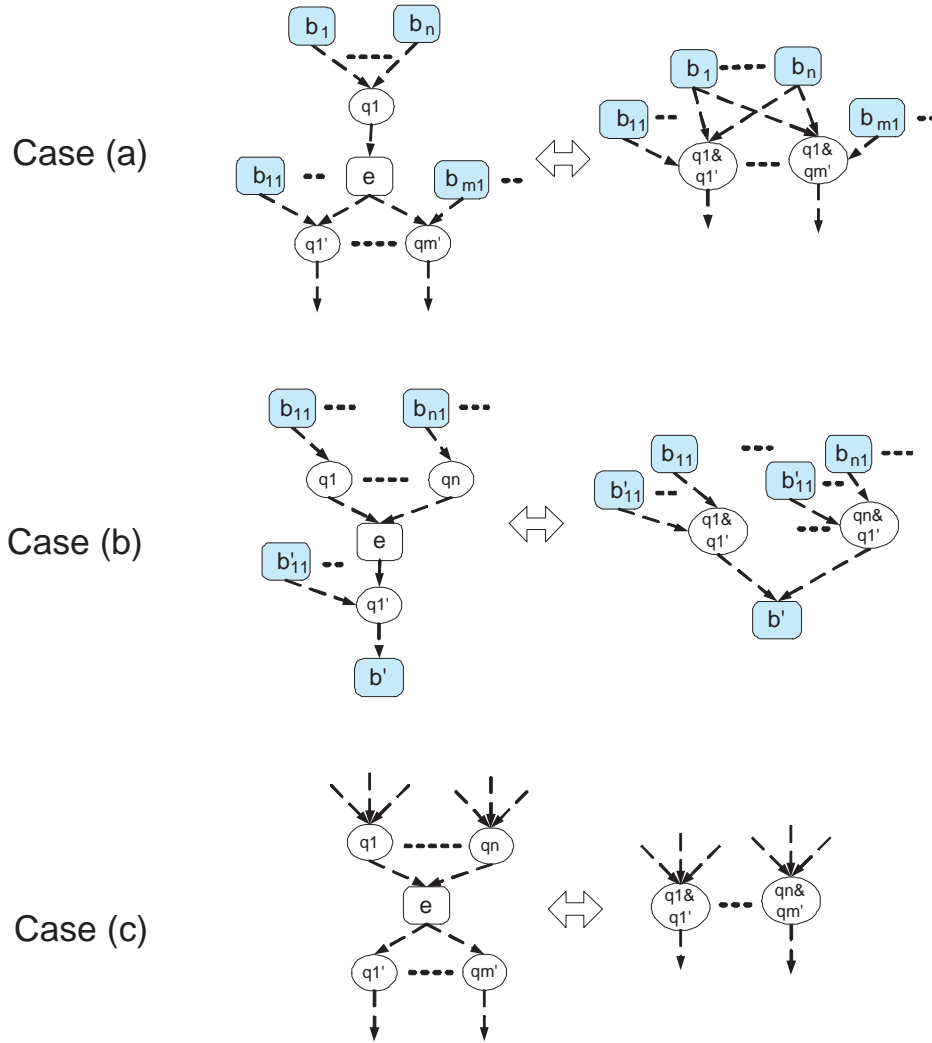


Figure 3.17: Identity elimination rule

merged with each outgoing control node to form  $m$  new control new control nodes. Each of the new control node conditions is the conjunction of the respective incoming and outgoing conditions (that is  $q_1 \wedge q'_i$  where  $1 \leq i \leq m$ ). The predecessors of the merged control node are derived by the union of predecessors of the respective incoming and outgoing nodes. The successor of the merged control node is the successor of the respective outgoing node.

The second case in which identity elimination may be applied is when we have  $n \geq 1$  control edges to  $e$  and only 1 outgoing control edges from  $e$ . After the transformation  $n$  new control nodes are formed by merging each incoming control node with the outgoing node  $q'_1$ . As

in the previous case, each of the new control node conditions is the conjunction of the respective incoming and outgoing conditions (that is  $q_i \wedge q'_1$  where  $1 \leq i \leq n$ ). The predecessors of the merged control node are derived by the union of predecessors of the respective incoming and the outgoing node  $q'_1$ . The successor of the merged control node is the successor of the  $q'_1$  as shown in Figure 3.17(Case c).

The third, and final case, of identity elimination rule is illustrated in Figure 3.17(Case c). Here, we have  $n > 1$  incoming control nodes and  $m > 1$  outgoing control nodes. The difference in this case is that the outgoing control nodes have  $e$  as their only predecessor. After the transformation, we get  $m \times n$  merged nodes, where each node has the control condition as the conjunction of the respective incoming and outgoing control condition (that is  $q_i \wedge q'_j$  where  $1 \leq i \leq n, 1 \leq j \leq m$ ).

In all the above cases, the transformation rule applies only if the new condition node in the transformed model will fire only if the respective incoming and outgoing nodes had fired in the original model, for the same execution scenario. This is ensured by restricting the identity elimination only if none of the variable that the incoming node depends on can be modified between the firing of incoming node and the outgoing node.

#### 3.5.4.1 Identity elimination rule

Recall that a control path from a behavior or control node to another behavior or control node in a given model,  $M$ , is the ordered set of behavior and control nodes representing the control path in the BCG of  $M$ . We will simply write a control path from behavior  $b$  to control node  $q$  as  $CP(b, q)$ . Note that there can be multiple control paths or no control path from  $b$  to  $q$ . We will assume that we are eliminating identity behavior  $e$  with  $n$  incoming control edges and  $m$  outgoing ones. If the following condition holds true,

$$(i) \quad \forall i, 1 \leq i \leq n, q_i \in Q, b' \in \mathcal{B}, q_i : \dots \& b' \& \dots \rightsquigarrow e \in b, \forall j, 1 \leq j \leq m, q'_j : \dots \& e \& \dots \rightsquigarrow b' \in b$$

$$(ii) \quad \nexists v \in \mathcal{V}, \text{ s.t. } e < out > \rightarrow v \in b$$

$$(iii) \quad \forall v \in \mathcal{V}, b_w \in \mathcal{B}, p_w \in \mathcal{P}, \text{ s.t. } q_i = f_b(\dots, v, \dots) \wedge b_w < p_w > \rightarrow v, \\ \nexists q'_j, \text{ s.t. } \exists CP(b_w, q'_j) \wedge q_i \notin CP(b_w, q'_j)$$

$$(iv) \quad \text{if } n > 1 \wedge m > 1, \forall q' \in Q, b', b'' \in \mathcal{B} \nexists q' : \dots \& b'' \& e \& \dots \rightsquigarrow b' \in b$$

then the identity elimination rule can be applied to  $e$  using the following syntactical manipulations

$$(i) \forall i, j, 1 \leq i \leq n, 1 \leq j \leq m, q_i : b_1 \& \dots \& b_n \rightsquigarrow e \in b \wedge q'_j : \dots \& e \& \dots \rightsquigarrow b' \in b, \\ b = b.((q_i \wedge q'_j) : \dots \& b_1 \& \dots \& b_n \& \dots \rightsquigarrow b')$$

$$(ii) \forall i, 1 \leq i \leq n, q_i : b_1 \& \dots \& b_n \rightsquigarrow e \in b, b = b - (q_i : b_1 \& \dots \& b_n \rightsquigarrow e)$$

$$(iii) \forall j, 1 \leq j \leq m, q'_j : \dots \& e \& \dots \rightsquigarrow b' \in b, b = b - (q'_j : \dots \& e \& \dots \rightsquigarrow b')$$

### 3.5.4.2 Soundness proof

According to the assumptions of the identity elimination rule, there does not exist any variable in the model such that  $e$  is writing to it. We will consider the soundness for the three different cases as shown in Figure 3.17.

In Case (a), consider the BCG for the models on LHS and RHS. On the LHS there is only one control node  $q_1$  such that there is a control edge from  $q_1$  to  $e$  in the model's BCG. Behavior nodes for  $b_1$  through  $b_n$  each have an edge to  $q_1$ . During execution of LHS,  $q_1$  fires when all of  $b_1$  through  $b_n$  have fired. Then  $q_1$  is evaluated. If it is true, then it fires, leading to the firing of  $e$  and the subsequent evaluation and possible firing of  $q'_1$  through  $q'_m$ . A node  $q'_j$ , where  $1 \leq j \leq m$  will fire only if  $q_1$  is true **and**  $q_m$  is also true. From assumption (iii), we know that none of the variables that  $q'_j$  depends on will be updated between the firing of  $q_1$  and  $q'_j$ . Therefore the evaluation of  $q'_j$  at the same time as the firing of  $q_1$  will produce the same result as firing of  $q'_j$  after  $e$ . Thus, the two evaluations can be replaced by the evaluation of  $q_1 \wedge q'_j$  before  $e$  fires. Generalizing for all control nodes  $q'_j, 1 \leq j \leq m$ , we get the same firing results as the model on RHS. Therefore, identity elimination rule is sound for Case (a).

Now consider the BCGs for the models on RHS and LHS of Case (b). On the LHS, the identity  $e$ , to be optimized, has control edges from  $n$  control nodes namely  $q_1$  through  $q_n$ . There is one control edge from  $e$  to node  $q'_1$ . During execution of the model on the LHS,  $e$  will fire if one of  $q_1$  through  $q_n$  fire. Upon firing,  $e$  enqueues a token in the queue for edge  $(e, q'_1)$ . If the other predecessors of  $q'_1$  also fire, then  $q'_1$  will be evaluated and may potentially fire, leading to

firing of  $b'$ . Without loss of generality, let us assume that  $q_i, 1 \leq i \leq n$  fires. That means that all predecessors of  $q_i$ , say  $b_{i1}$  through  $b_{ik}$ , must fire. Again, using assumption (iii), the evaluation of  $q'_1$  will yield the same result if it fired at the same time as  $q_i$ . Therefore, if  $q_i \wedge q'_1$  is true (at the time of  $q_i$ 's firing, then  $b'$  will fire, otherwise  $b'$  will not fire. Now, generalizing our result for all  $i, 1 \leq i \leq n$ , we deduce that the execution of LHS and RHS models will have the same set of possible traces for all corresponding observed variables. This is because  $e$  does not write to any variable and the firing results for all non-identity behaviors have been shown to be identical. Therefore, identity elimination rule is sound for Case (b).

Finally, we look at the BCGs of models on the LHS and RHS of Case (c). The identity behavior  $e$ , that will be eliminated, has  $n$  incoming control edges and  $m$  outgoing control edges to nodes  $q_1, \dots, q_n$  and  $q'_1, q'_2, \dots, q'_m$ , respectively. Consider control node  $q_i$ , where  $1 \leq i \leq n$  and control node  $q'_j$ , where  $1 \leq j \leq m$ , such that edges  $(q_i, e)$  and  $(e, q'_j)$  exist in the BCG of the model on the LHS. Let  $b'_j$  be a behavior node such that edge  $(q'_j, b'_j)$  also exists in the BCG of LHS model. Let  $b_{i1}$  through  $b_{ik}$  be all the behaviors with edge to  $q_i$ . From assumption (iv), we know that there are no behaviors, besides  $e$  that may have an edge to  $q'_j$ . Without loss of generality, we assume that during model execution,  $b_{i1}$  through  $b_{ik}$  fire resulting in the evaluation of  $q_i$ . If  $q_i$  is true,  $e$  fires leading to evaluation of  $q'_j$ . If  $q'_j$  also evaluates to true, then  $b'_j$  will also fire. Again, from assumption (iii), we know that the evaluation result of  $q'_j$  will not be affected, if it were to fire at the same time as  $q_i$ . Hence, we deduce that  $b'_j$  will fire if  $b_{i1}$  through  $b_{ik}$  fire and  $q_i \wedge q'_j$  evaluates to true. Generalizing our result for all  $i, 1 \leq i \leq n$  and for all  $j, 1 \leq j \leq m$ , we deduce that the execution results will be identical for models on LHS and RHS. This is because  $e$  does not write to any variable and the firing results of all non-identity behaviors are shown to be identical for the same initialization. Therefore, identity elimination rule is sound for Case (c).

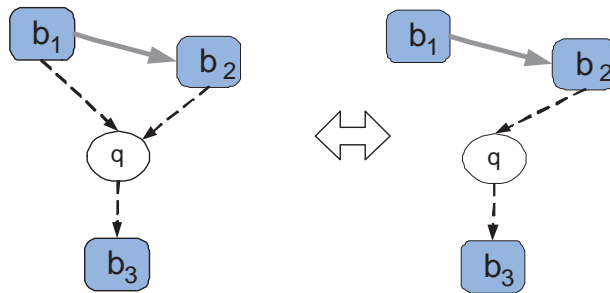


Figure 3.18: Redundant control dependency elimination

### 3.5.5 Redundant control dependency elimination

If a control flow relation has more than one predecessors, where one predecessor dominates another, then the relation can be simplified using redundant control dependency elimination (RCDE) rule. Consider the firing semantics of the BCG for the example model shown in LHS of Figure 3.18. The dominator relation between  $b_1$  and  $b_2$  ( $b_1 \triangleright b_2$ ) is shown by a solid grey arrow from  $b_1$  to  $b_2$ . Both behaviors  $b_1$  and  $b_2$  have an edge to control node  $q$ . According to the semantics of BCG,  $q$  has two queues, one for each edge from the behaviors. If  $b_1 \triangleright b_2$ , then the firing of  $b_2$  indicates that a unique firing of  $b_1$  has already taken place. Therefore, if a token is enqueued for  $b_2$ , then a token has already been enqueued for  $b_1$ . The model on RHS of Figure 3.18 is derived by removing the control edge from  $b_1$  to  $q$ . The transformed model would execute just like the original model, since the edge from  $b_1$  to  $q$  was redundant due to the presence of edge from  $b_2$  to  $q$ .

#### 3.5.5.1 RCDE rule

The RCDE rule can simply be written as

If  $\exists q \in Q, b_1, b_2, b_3, b \in \mathcal{B}$ , s.t.  $q : \dots \& b_1 \& b_2 \& \dots \rightsquigarrow b_3 \in b \wedge b_1 \triangleright b_2$ , then

$$b = (b - q : \dots \& b_1 \& b_2 \& \dots \rightsquigarrow b_3). q : \dots \& b_2 \& \dots \rightsquigarrow b_3$$

#### 3.5.5.2 Soundness proof

The soundness proof of RCDE is fairly straightforward. Consider the model shown on the LHS of Figure 3.18. We know that  $b_1 \triangleright b_2$  from the assumption for RCDE. Consider the BCG for the model on LHS. The control node  $q$  will fire only if both  $b_1$  and  $b_2$  have fired, according to model execution semantics. Now, by the dominator definition, we know that any firing of  $b_2$  implies that  $b_1$  must have fired at least once in the past before the previous firing of  $b_2$  or *vsp*. Therefore, if  $b_2$  fires and enqueues a token for the queue of edge  $(b_2, q)$ , then we are assured that a token is already enqueued in the queue for the edge  $(b_1, q)$ . By the model execution semantics,  $q$  attempts to dequeue a token from each of its incoming edge queues before evaluation the control condition and potentially firing. If a token is found in queue for  $(b_2, q)$ , then we can skip the

checking of queue for  $(b_1, q)$ . If a token is not found in queue for  $(b_2, q)$  (that is  $b_2$  does not fire), then  $q$  will not prepare to fire in any scenario. Therefore, the edge  $(b_1, q)$  is redundant for model execution. Hence, the models on LHS and RHS have the same execution results. Thus, we have proved that redundant control dependency elimination rule is sound.

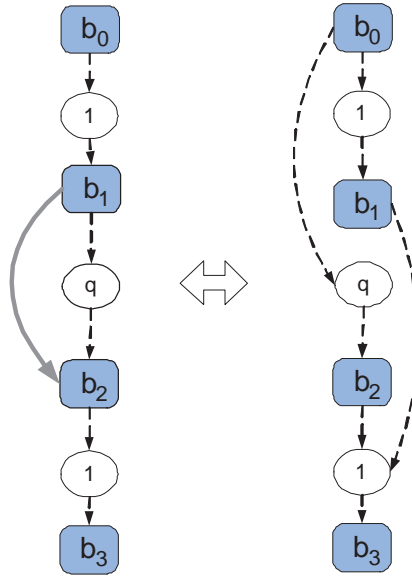


Figure 3.19: Control relaxation rule

### 3.5.6 Control relaxation

If there is a control dependency between two behaviors without any data dependency, then such a dependency may be removed using the control relaxation rule. Figure 3.19 illustrates the control relaxation rule more specifically. On the LHS, we have a model with a control dependency from behavior  $b_1$  to  $b_2$  with condition  $q$ . Also  $b_1$  dominates  $b_2$ , as shown by the solid grey arrow from  $b_1$  to  $b_2$ . If there is no data dependency from  $b_1$  to either  $q$  or  $b_2$ , then  $q$  and  $b_2$  may fire *in parallel* with  $b_1$  without creating any difference in the value trace of observed variables. Specifically, if there does not exist any variable  $v$  in the model such that  $b_1$  writes to  $v$  **and** either  $b_2$  reads  $v$  or  $q$  is a boolean function of  $v$ , then the dependency from  $b_1$  to  $b_2$  may be replaced by two new dependencies, namely one from  $b_0$  to  $b_2$  and another from  $b_1$  to  $b_3$ . The result of this transformation is that  $b_2$  does not need to wait for  $b_1$  to fire before firing itself. Essentially,  $b_2$  may now fire *in parallel* with  $b_1$ .

### 3.5.6.1 Control relaxation rule

The control relaxation transformation can be applied if the following conditions hold true. If

- (i)  $\exists b_1, b_2, b \in \mathcal{B}, q \in \mathcal{Q}$ , s.t.  $q : \dots \& b_1 \& \dots \rightsquigarrow b_2 \in b \wedge b_1 \triangleright b_2$
- (ii)  $\exists b_0 \in \mathcal{B}$ , s.t.  $1 : b_0 \rightsquigarrow b_1 \in b \wedge (\nexists b' (\neq b_0) \in \mathcal{B}, \nexists q' \in \mathcal{Q}$  s.t.  $q' : \dots \& b' \& \dots \rightsquigarrow b_1 \in b)$
- (iii)  $\exists b_3 \in \mathcal{B}$ , s.t.  $1 : b_2 \rightsquigarrow b_3 \in b \wedge (\nexists b' (\neq b_3) \in \mathcal{B}, \nexists q' \in \mathcal{Q}$  s.t.  $q' : \dots \& b_2 \& \dots \rightsquigarrow b' \in b)$
- (iv)  $\nexists v \in \mathcal{V}, p_1, p_2 \in \mathcal{P}$ , s.t.  $(b_1 < p_1 > \rightarrow v \in b) \wedge (q = f(\dots v \dots) \vee (v \rightarrow b_2 < p_2 > \in b))$

then, the following syntactical manipulations may be performed

- (i)  $\forall i, 1 \leq i \leq n, b_1^i \in \mathcal{B}$ , s.t.  $q : \dots \& b_1^i \& b_1 \& b_1^{i+1} \& \dots \& b_1^n \rightsquigarrow b_2 \in b$ ,  
 $b = (b - q : \dots \& b_1^i \& b_1 \& b_1^{i+1} \& \dots \& b_1^n \rightsquigarrow b_2). q : \dots \& b_1^i \& b_0 \& b_1^{i+1} \& \dots \& b_1^n \rightsquigarrow b_2)$
- (ii)  $b = (b - 1 : b_2 \rightsquigarrow b_3). 1 : b_1 \& b_2 \rightsquigarrow b_3$

### 3.5.6.2 Soundness proof

The soundness proof for control relaxation rule is as follows. Consider the LHS and RHS models in Figure 3.19. The only path to  $b_1$  is from  $b_0$ , therefore, only the firing of  $b_0$  will trigger the firing of  $b_1$ . As per the assumptions,  $q$  and  $b_2$  do not have any data dependencies on  $b_1$ . Let there be a variable  $v$  in the model on LHS, such that

$$\exists p_2 \in \mathcal{P}, v \rightarrow b_2 < p_2 > \in b$$

By assumption (iv), we know that

$$\nexists p_1 \in \mathcal{P}, \text{ s.t. } b_1 < p_1 > \rightarrow v \in b$$

Also,  $q$  does not depend on any variable written by  $b_1$ . Therefore, if  $q$  is evaluated either after  $b_1$  of before it, the evaluation result will be the same. Assume that in the BCG of LHS, the edge  $(b_1, q)$  is replaced with edge  $(b_0, q)$ . Since  $b_0$  fires once for every firing of  $b_1$ , the pattern of incoming tokens to queues of  $q$  will remain unchanged. Thus  $b_2$  will fire as before and with same results, since  $b_1$  will not modify any inputs of  $b_2$ .

Now consider the control node following  $b_2$ , with control condition 1. As per assumption (iii), there are no other control nodes with an edge from  $b_2$ . Therefore, for every firing of  $b_2$ , the successor node  $b_3$  will also fire once. In the LHS model,  $b_3$  would fire if both  $b_1$  and  $b_2$  would fire, since  $b_2 \triangleright b_3$ . In the model on the RHS,  $b_3$  fires if both  $b_1$  and  $b_2$  fire. This is ensured by the extra control dependency added as an edge from  $b_1$  to 1. Therefore, the execution of both LHS and RHS will produce a different firing order of behaviors, but the same value traces for all observed variables. Hence, the control relaxation rule is sound.

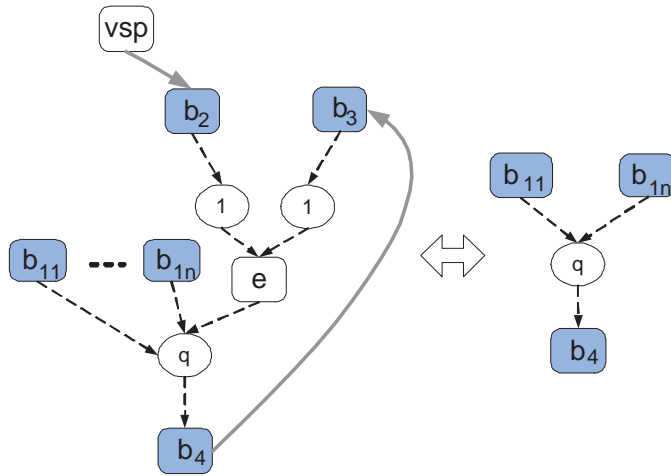


Figure 3.20: Streamlining rule

### 3.5.7 Streamlining

The streamlining rule is used to simplify the model algebraic expression by removing redundant control dependencies. It is similar to control dependency elimination and identity elimination and can be used in cases when neither of the above rules can be applied. As illustrated in Figure 3.20, streamlining rule can be applied for a given control node  $q$  with successor  $b_4$  and  $n + 1$  predecessors, where  $n \geq 1$  one of the predecessors is an identity behavior  $e$ . Identity  $e$ , in turn has two control dependencies (both with condition 1), from two behaviors, say  $b_2$  and  $b_3$ . Behavior  $b_2$  is either  $VSP$  of the model or dominated by  $VSP$ . Behavior  $b_3$  is either  $b_4$  or dominated by  $b_4$ . If  $b_3 \neq b_4$ , then  $b_3$  must fire at least once *between* any two firings of  $b_4$ . If all the conditions are met, then the model may be transformed by removing the control dependency from  $e$  to  $b_4$ , effectively deleting the edge from  $e$  to  $q$ .

### 3.5.7.1 Streamlining rule

The streamlining transformation can be applied if the following conditions hold true. If

- (i)  $\forall n \geq 1, \exists b_{11}, b_{12}, \dots, b_{1n}, b_2, b_3, b_4, e, b \in \mathcal{B}, q \in \mathcal{Q}$ , s.t.  
 $q : b_{11} \& \dots \& b_{1n} \& e \rightsquigarrow b_4 \in b \wedge 1 : b_2 \rightsquigarrow e \in b \wedge 1 : b_3 \rightsquigarrow e \in b$
- (ii)  $(b_2 = vsp \vee vsp \triangleright b_2) \wedge (b_3 = b_4 \vee (b_4 \triangleright b_3 \wedge b_3 \triangleright (b_4, b_4)))$

then, the following syntactical manipulation may be performed

$$b = (b - q : b_{11} \& \dots \& b_{1n} \& e \rightsquigarrow b_4). q : b_{11} \& \dots \& b_{1n} \rightsquigarrow b_4$$

### 3.5.7.2 Soundness proof

The soundness proof for streamlining rule is performed by comparing the BCGs for the models on the LHS and the RHS of Figure 3.20. In the model on the LHS, we have a control node  $q$  with control edges from behaviors  $b_{11}$  through  $b_{1n}$  and identity  $e$ . Also,  $q$  has a control edge to behavior  $b_4$ . Now, from the execution semantics  $b_4$  will fire if all of  $b_{11}$  through  $b_{1n}$  and identity  $e$  fire and then condition  $q$  evaluates to true. Let us consider the firing of identity  $e$ . Node  $e$  has two incoming control paths from  $b_1$  and  $b_2$ . Therefore,  $e$  will fire if and only if either  $b_1$  or  $b_2$  fire. From assumption (ii),  $b_1$  is either  $vsp$  or dominated by it. Therefore, during all of model execution  $b_1$  will fire only once, since  $vsp$  fires only once for any model execution. Now, by the second part of assumption (ii),  $b_2$  is either same as  $b_4$  or fires once for each firing of  $b_4$  (after  $b_4$  has fired). Therefore, the first firing of  $e$  will result from the control path from  $b_1$ , leading to a potential firing of  $b_4$ , if all of  $b_{11}$  through  $b_{1n}$  fire. If  $b_4$  fires, then  $b_2$  will also fire, leading to another firing of  $e$  and a subsequent potential firing of  $b_4$  if all of  $b_{11}$  through  $b_{1n}$  fire. Therefore, every firing of  $b_4$  is dependent only on the firing of  $b_{11}$  through  $b_{1n}$  and the evaluation of  $q$  to be true. Thus, the control dependency represented by the edge  $(e, q)$  is redundant. Hence, the models on both LHS and RHS produce the same firing of behaviors. Since, no data relations are modified, we have proven the soundness of streamlining rule.

## 3.6 Chapter Summary

In this chapter, we introduced a formalism called Model Algebra, which can be used for functional verification of system level models. The objects and composition rules of Model Algebra allowed us to represent hierarchical system level models as expressions. We then presented the formal execution semantics of model algebraic descriptions using behavior control graphs. We also established a notion of functional equivalence of two models based on the value trace of variables in the models. This led us to define functionality preserving transformation rules on model algebraic descriptions. The soundness of these transformation rules was proved based on our notion of equivalence. The expressive power and well defined rules in MA can be used to derive new equivalent models from the specification and perform correct transformations on them. The formalization of models using Model Algebra shows promise of significant impact on system level verification.

## Chapter 4

# Functionality Preserving Refinements

In the last chapter we proposed a formalism for representing system level models as expression. Within this formalism, called Model Algebra, we also defined a set of correct transformation rules that can be used to derive an equivalent model from another model. In this chapter, we will apply the established theory of Model Algebra to a real life system design methodology.

The design methodology is based on the SpecC methodology that starts with an executable functional specification model of the design and is gradually refined into a cycle accurate model that is ready to be manufactured using traditional design automation tools. The refinement process consists of several steps. Each of these steps syntactically manipulates the model either by rearranging or replacing objects in the model. We will identify the key refinement steps that can be verified in Model Algebra and give algorithms for implementing such refinements. Proofs of correctness for each refinement algorithm is based on the previously proven transformation rules. These proofs will establish the correctness of the refinement algorithms and ensure that the derived models are indeed functionally equivalent to the original models.

### 4.1 Refinement Based Design Methodology

A possible system level design methodology is illustrated in Figure 4.1. The designer starts with a specification of the platform architecture in terms of components and system busses and their connections. The functional specification is provided as an executable model. This specification model forms the golden reference for all models in upcoming design steps. Based

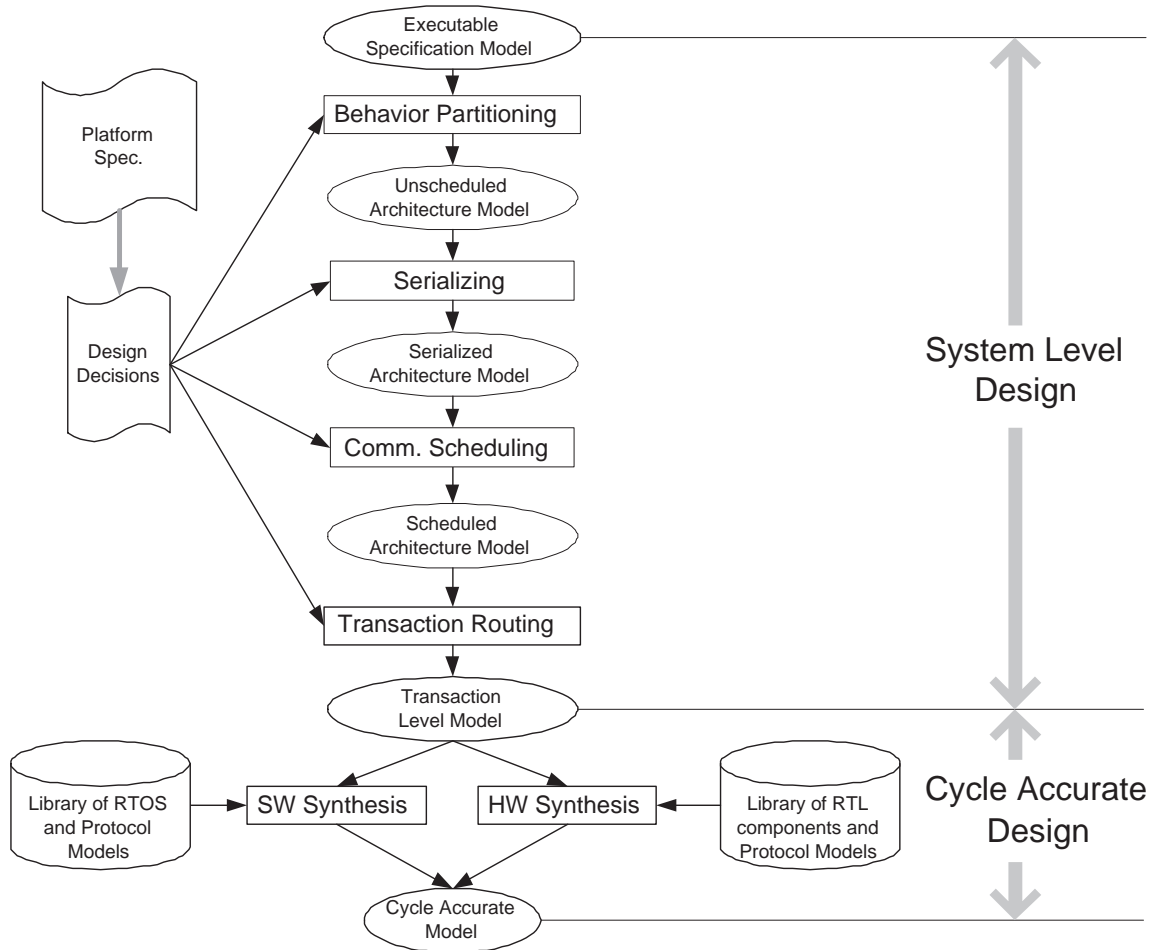


Figure 4.1: Refinement based design methodology

on the platform model and certain performance constraints for the system, the designer makes decisions to refine the specification model in a step wise fashion. At each step, a model is changed to reflect features of the platform, resulting in a new executable model, which can be further refined.

The process of going from the purely functional specification model to a very detailed cycle accurate model, that can be passed on to traditional design automation tools, is broadly divided into two major design steps. The first is the system level design step that takes the functional specification to a transaction level model. The transaction level model contains details pertaining to the mapping of functional tasks to appropriate components and the routing of abstract communication over system busses. However, it does not have any information on the micro-architectural

details of the components and the busses themselves.

The transaction level model is handed off to the next step of cycle accurate (CA) design. During CA design, the functional task mapped to hardware components undergo HW synthesis to derive RTL models of the application and the bus interface. Similarly, for task mapped to processors, we perform OS targeting and compilation to get a SW binary that can be down loaded to the processor and simulated using the instruction set simulator of the processor.

Our work falls under the domain of system level design. Hence, in this dissertation, we will concern ourselves with models and transformations that are encountered in going from functional specification to transaction level. We will discuss four key refinements and their verification in Model Algebra. These refinements are

1. *Behavior Partitioning*: During this refinement, the hierarchy of behaviors in the functional model is rearranged to reflect the mapping of leaf level behaviors to component behaviors [PAG02]. New transaction links may be introduced to allow data transfer between behaviors mapped to different components or to synchronize between such behaviors. The refinement is driven by the design decisions including mapping of behaviors to components and mapping of new transaction links to busses.
2. *Serializing*: In order to prepare the model for HW synthesis, behaviors that must be executed with a single controller are serialized. The serialization refinement converts parallel composition of relevant behaviors into sequential compositions.
3. *Communication scheduling*: The performance of a design may be improved by modifying the scheduling of bus transactions [MDR<sup>+</sup>00]. The corresponding refinement process reorders the communicating behaviors to reflect the scheduling decision.
4. *Transaction routing*: Transaction links may be reassigned from one bus to another to improve performance. Sometimes, due to interface incompatibility, a bridge or router may be inserted between communicating components [BK87]. This results in the splitting of a transaction link (between such components) into two links, one from the sender to the router, and another from the router to the receiver. Functionally, the router is simply an identity behavior that loops forever, reading from one link and writing to another.

We will now look at each of the design steps in greater detail. Refinement algorithms for deriving new models for each set of design decisions will also be presented and proved using rules of model algebra.

## 4.2 Behavior Partitioning

A given specification model may consist of an arbitrary hierarchy of behaviors. During partitioning, we determine the number of processing elements (PEs) that will be needed to implement the design. The leaf behaviors in the specification are then distributed over these PEs. The PEs are assumed to execute concurrently. Thus, in this step, the design decision is to map each leaf behavior in the specification model to a PE. We will consider two different refinement algorithms that create a different style of architecture model for the same design decisions. The styles differ in the size of the models and the amount of synchronization required between PEs. However, both algorithms will be proven to produce models that are functionally equivalent to the specification.

### 4.2.1 Design Decisions

The decision to implement a given task on a certain PE is determined by several factors. Often, restrictions on the type of PE limit the choice of mapping. For example, a task involving floating point operations is better implemented on a processor with a built in floating point unit. Although the floating point operation may be emulated in software on a processor without a dedicated floating point unit, the performance of an emulated implementation is significantly poorer. Another motivation for using multiple PEs may be to utilize the built in parallelism in the application in order to improve performance. Furthermore, most hard real time applications may require a mix of SW processors and specialized HW components to meet both flexibility and performance demands.

We now show how the design decisions are input to the refinement algorithm. Let  $M_s$  be the specification model that needs to be refined. Let  $Leafs(M_s)$  be the set of all leaf level behaviors in  $M_s$ . Let there be  $n$  different PEs in the platform architecture named  $PE_1$  through  $PE_n$ . Let  $map_i$  be the set of leaf behaviors in  $M_s$  that are assigned for implementation on component  $PE_i$ , where  $1 \leq i \leq n$ . The mapping is such that

1. A leaf behavior is mapped to only one PE, i.e.  $\forall i, j, 1 \leq i, j \leq n, i \neq j, map_i \cap map_j = \{\}$
2. All leaf behaviors are mapped to some PE, i.e.  $\bigcup_{i=1}^n map_i = Leafs(M_s)$

The input to the refinement algorithm from the designer is  $n, map_i$ , where  $1 \leq i \leq n$ .

#### 4.2.2 Refinement Algorithm for Lock-Step output

The input to behavior partition refinement is the original specification model and the design decision of mapping leaf level behaviors to the PEs. The goal is to generate a model that represents the mapping of system functionality to architecture components. The specification model is an arbitrary hierarchy of behaviors representing the system functionality. Model refinement would distribute the behaviors onto components that run concurrently in the system. It must be noted that refinement does not in any way influence the mapping decision. The designer is free to choose any mapping of behaviors to components and refinement would produce a model that represents it. However, each leaf behavior in the specification model must be mapped to only one component.

Figure 4.2 shows the application of the lock-step refinement algorithm on a behavior  $b_i$  of a given specification model  $M_s$ . The rule shown in the figure is applied generally to all the behaviors in the specification. We assume that the architecture consists of two processing elements PE1 and PE2. Let  $map_1$  be the set of leaf level behaviors of  $M_s$  that are mapped to PE1 and  $map_2$  be the set of leaf behaviors mapped to PE2. In the illustration, we have assumed that  $b_i \in map_1$ .

The architecture model  $M_a$  is created as follows. We start by creating a parallel composition of all PEs, to denote the concurrent execution of PEs in the system. We have,

$$M_a = [PE_1].[PE_2]...[PE_n].1 : vsp \rightsquigarrow PE_1...1 : vsp \rightsquigarrow PE_n.1 : PE_1...PE_n \rightsquigarrow vtp$$

Then, we copy the specification model inside each PE. The leaf level behaviors in each copy are replaced by special hierarchical behaviors depending on the mapping. We will demonstrate this replacement using the illustration in Figure 4.2. Since  $b_i$  is mapped to PE1, in the architecture model  $M_a$ , we will replace copy of  $b_i$  inside PE1 with  $b_{i1}$ , where hierarchical behavior  $b_{i1}$  is created as follows.

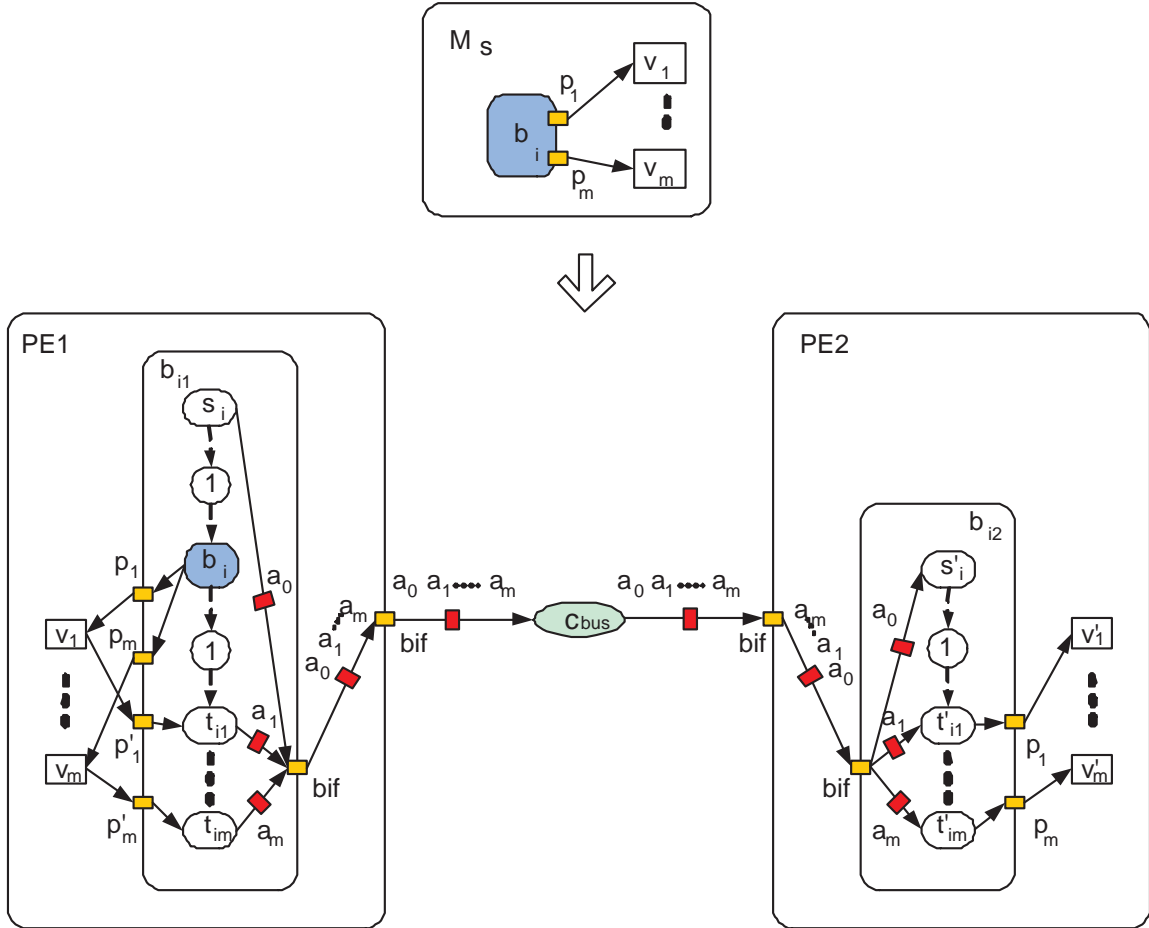


Figure 4.2: Lock-step style output of behavior partitioning refinement

Let  $\bar{M}_s$  be the model  $M_s$  after flattening down to leaf behaviors. Let there be variables  $v_1$  through  $v_m$ , such that  $b_i$  is a writer for all these variables. That is

$$\forall j, 1 \leq j \leq m, \exists b_i < p_j > \rightarrow v_j \in \bar{M}_s$$

The hierarchical behavior  $b_{i1}$  is a sequential composition of  $m+2$  behaviors namely  $s_i, b_i, t_{i1}, \dots, t_{im}$ , where all these behaviors, except  $b_i$ , are identity. We also copy the ports of  $b_i$  onto  $b_{i1}$ , such that port  $p$  on  $b_{i1}$  is mapped to port  $p$  on  $b_i$ . Additionally, we create  $m$  new in ports on  $b_{i1}$ , that allow behaviors  $t_1$  through  $t_m$  to read the local copies of variables  $v_1$  through  $v_m$ . We also create an extra port  $bif$  to work as the bus interface, which is mapped to channel  $c_{bus}$  that represents the system bus. Blocking write relations to  $bif$  are added inside  $b_{i1}$  to synchronize its execution with other PEs.

If a leaf behavior is not mapped to a PE, then its local copy inside the PE is replaced by a special hierarchical behavior as follows. We will demonstrate this case on the replacement of  $b_i$  inside PE2, since  $b_i \notin \text{map}_2$ . Let  $b_i$  be replaced by  $b_{i2}$  inside PE2 as follows. We create a sequential composition of  $m + 1$  identity behaviors namely  $s'_i, t'_{i1}, \dots, t'_{im}$ . We also create  $m$  in ports on  $b_{i2}$  corresponding to the  $m$  out ports of  $b_i$  in  $M_s$ . The ports are mapped to the local copies of the  $m$  variables that are written by  $b_i$  in  $M_s$ . Inside  $b_{i2}$ , the ports are mapped to the outputs of the identity behaviors  $t_1$  through  $t_m$ . We also create an addition *bif* port as in the previous case. Blocking read relations are created using *bif* to all the identity behaviors inside  $b_{i2}$ . The addressing of these relations is done such that  $s_i$  sends a transaction to  $s'_i$  labeled  $a_0$ . Transactions for sending variables written by  $b_i$  to update the local copies in PE2 are created such that  $\forall j, 1 \leq j \leq m, t_{ij}$  sends variable  $v_j$  to  $t'_{ij}$  using transaction addressed  $a_j$  over  $c_{bus}$ .

The idea of the lock-step output is that each leaf behavior executes inside the PE it was mapped to. Any outputs updated by the leaf behavior execution are immediately copied to the local copies of the outputs inside other PEs. Behaviors following the execution of this leaf behavior must wait until all the copies everywhere in the system are updated. Therefore, the state of the system at leaf behavior boundaries is preserved in the architecture model.

The pseudo code for the lock-step style refinement is presented in Algorithm 2. The algorithm uses a method called *Encapsulate* that is presented in Algorithm 1. The *Encapsulate* method is the inverse of flattening a hierarchical behavior with a single child behaviors.

### 4.2.3 Correctness Proof for Lock-Step Refinement

We now present the proof of correctness for the refinement algorithm for generating lock-step style architecture model. Let  $M_s$  be the specification model and let  $b_i$  be a leaf level behavior in  $M_s$ . Assume that the designer maps  $b_i$  to  $PE_k$ . In other words,  $b_i \in \text{map}_k$ . Figure 4.3 shows the basic proof steps we follow. Our proof strategy is to take the generated architecture model and use the functionality preserving transformation rules to derive the specification. The illustration is for an architecture with two PEs, but the proof can be easily generalized to multiple PEs.

We begin by using the flattening transformation rule to create a flattened architecture model. Consider behavior  $b_i$  in the flattened architecture model as shown in Figure 4.3(a). By the

---

**Algorithm 1** Encapsulte (Behavior b, Behavior Parent)

---

1:  $b^e = [b].1 : vsp^e \rightsquigarrow b.1 : b \rightsquigarrow vt p^e$   
2: **if**  $\forall v \in \mathcal{V}, p \in \mathcal{P}, v \rightarrow b \langle p \rangle \in \text{Parent}$  **then**  
3:    $b^e = b^e.I \langle p \rangle \rightarrow b \langle p \rangle$   
4: **end if**  
5: **if**  $\forall p, p' \in \mathcal{P}, I \langle p' \rangle \rightarrow b \langle p \rangle \in \text{Parent}$  **then**  
6:    $b^e = b^e.I \langle p \rangle \rightarrow b \langle p \rangle$   
7: **end if**  
8: **if**  $\forall v \in \mathcal{V}, p \in \mathcal{P}, b \langle p \rangle \rightarrow v \in \text{Parent}$  **then**  
9:    $b^e = b^e.b \langle p \rangle \rightarrow I \langle p \rangle$   
10: **end if**  
11: **if**  $\forall p, p' \in \mathcal{P}, b \langle p \rangle \rightarrow I \langle p' \rangle \in \text{Parent}$  **then**  
12:    $b^e = b^e.b \langle p \rangle \rightarrow I \langle p \rangle$   
13: **end if**  
14: **if**  $\forall p, p' \in \mathcal{P}, a \in \mathcal{A}, a : I \langle p' \rangle \mapsto b \langle p \rangle \in \text{Parent}$  **then**  
15:    $b^e = b^e.a : I \langle p \rangle \mapsto b \langle p \rangle$   
16: **end if**  
17: **if**  $\forall p, p' \in \mathcal{P}, a \in \mathcal{A}, a : b \langle p \rangle \rightarrow I \langle p' \rangle \in \text{Parent}$  **then**  
18:    $b^e = b^e.a : b \langle p \rangle \rightarrow I \langle p \rangle$   
19: **end if**  
20: **if**  $\forall p, p' \in \mathcal{P}, a \in \mathcal{A}, c \in \mathcal{C}, b' \in \mathcal{B}, c \langle a \rangle : b' \langle p' \rangle \mapsto b \langle p \rangle \in \text{Parent}$  **then**  
21:    $b^e = b^e.a : I \langle p \rangle \mapsto b \langle p \rangle$   
22: **end if**  
23: **if**  $\forall p, p' \in \mathcal{P}, a \in \mathcal{A}, c \in \mathcal{C}, b' \in \mathcal{B} c \langle a \rangle : b \langle p \rangle \rightarrow b' \langle p' \rangle \in \text{Parent}$  **then**  
24:    $b^e = b^e.a : b \langle p \rangle \rightarrow I \langle p \rangle$   
25: **end if**  
26:  $\text{Parent} = \text{Parent}|_{b=b^e}$ 

---

---

**Algorithm 2** Refine2LockStep ( $M_s, n, map_1, \dots, map_n$ )

---

```
1:  $\bar{M}_s = flatten(M_s); Leafs(\bar{M}_s) = \{b_1, b_2, \dots\}$ 
2: for  $i = 1$  TO  $n$  do
3:    $PE_i = \bar{M}_s; \forall v \in vars(\bar{M}_s), PE_i = PE_i|_{v=v_i}; \forall q \in ctrls(\bar{M}_s), q_i = q|_{v=v_i}, PE_i = PE_i|_{q=q_i}$ 
4:   for  $j = 1$  TO  $|Leafs(\bar{M}_s)|$  do
5:      $vars_j = \{v | b_j < p \rangle \rightarrow v \in \bar{M}_s \wedge (\exists q, q = f(\dots, v, \dots) \vee \exists b \notin map_i, v \rightarrow b < p' \rangle \in \bar{M}_s)$ 
6:     if  $b_j \in map_i$  then
7:        $b_j^i = Encapsulate(b_j, PE_i)$ 
8:        $b_j^i = (b_j^i - 1 : vsp_j^i \rightsquigarrow b_j^i - 1 : b_j \rightsquigarrow vtp_j^i).1 : vsp_j^i \rightsquigarrow s_j^i$ 
9:        $b_j^i = b_j^i.a_0^j : s_j^i < out \rangle \mapsto I < p_0 \rangle .1 : b_j \rightsquigarrow t_{j1}^i$ 
10:      for  $k = 1$  TO  $|vars_j|$  do
11:         $b_j^i = b_j^i.b_j < p_k \rangle \rightarrow I < p_k \rangle .I < p_k' \rangle \rightarrow t_{jk}^i < in \rangle$ 
12:         $b_j^i = b_j^i.a_k^j : t_{jk}^i < out \rangle \mapsto I < p_0 \rangle .1 : t_{jk}^i \rightsquigarrow t_{j(k+1)}^i$ 
13:      end for
14:       $b_j^i = (b_j^i - 1 : t_{j|vars_j|}^i \rightsquigarrow t_{j(|vars_j|+1)}^i).1 : t_{j|vars_j|}^i \rightsquigarrow vtp_j^i$ 
15:      else
16:         $b_j^i = 1 : vsp_j^i \rightsquigarrow s_j^i.a_o^j : I < p_0 \rangle \mapsto s_j^i < in \rangle$ 
17:         $last = s_j^i$ 
18:        for  $k = 1$  TO  $|vars_j|$  do
19:          if  $(\exists q, q = f(\dots, vars_j(k), \dots)) \vee (\exists b' \in map_i, vars_j(k) \rightarrow b' < p' \rangle \in \bar{M}_s)$  then
20:             $b_j^i = b_j^i.a_k^j : I < p_k \rangle \mapsto t_{jk}^i < in \rangle .t_{jk}^i < out \rangle \rightarrow I < p_k \rangle .1 : last \rightsquigarrow t_{jk}^i; last = t_{jk}^i$ 
21:          end if
22:        end for
23:         $b_j^i = b_j^i.1 : last \rightsquigarrow vtp_j^i$ 
24:        end if
25:         $PE_i = PE_i|_{b_j=b_j^i}$ 
26:      end for
27:       $M_a = M_a.[PE_i].vsp \rightsquigarrow PE_i$ 
28:    end for
29:   $M_a = M_a.PE_1 \& PE_2 \& \dots \& PE_n \rightsquigarrow vtp$ 
```

---

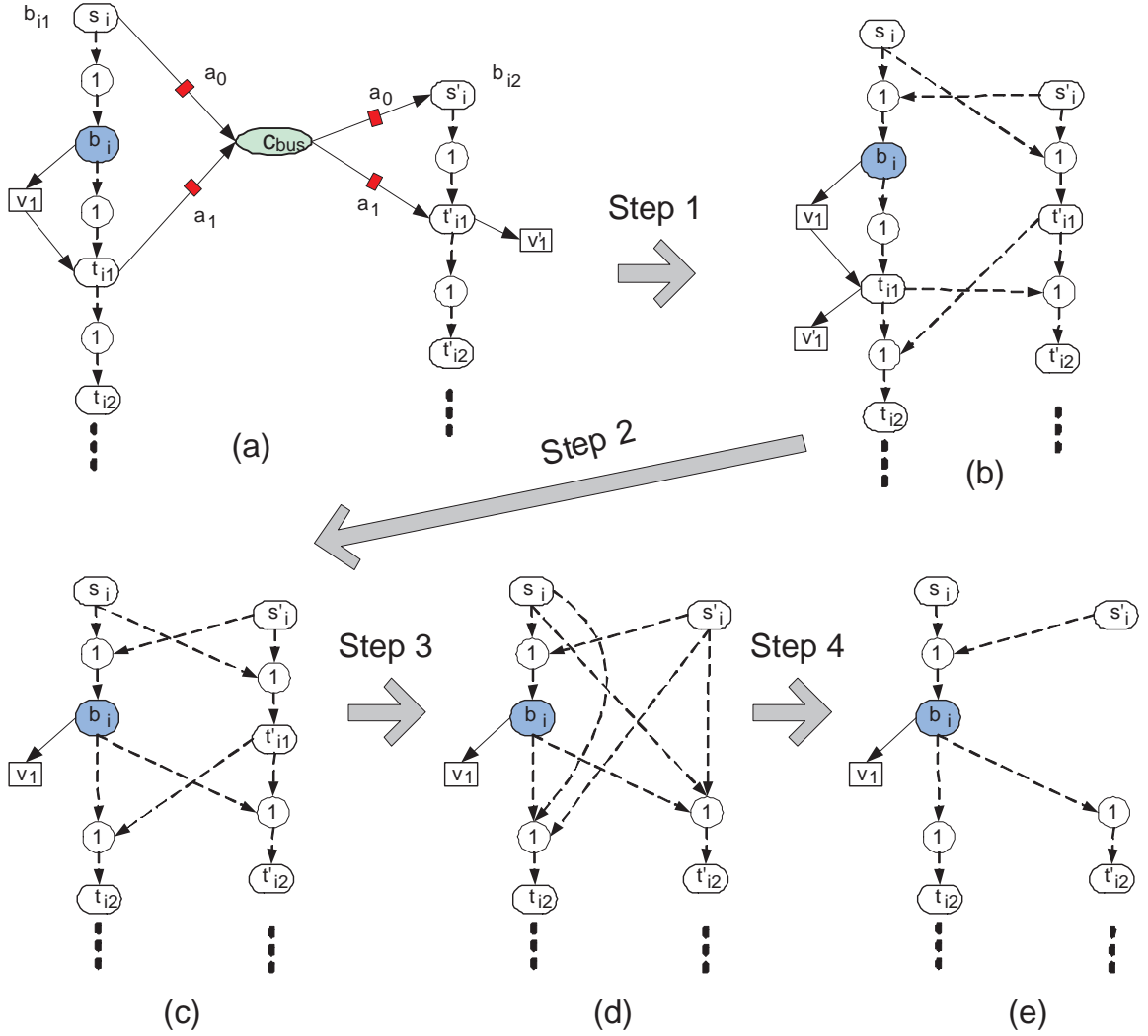


Figure 4.3: Proof steps for refinement to lock-step style model

refinement results shown in Section 4.2.2, we have channel transactions from  $s_i$  to  $s'_i$ , from  $t'_{i1}$  to  $t_{i1}$  and so on.

We apply the transaction link resolution rule to convert transaction links into new control and data dependencies (Step 1). In this context, the link

$$c \langle a_0 \rangle : s_i \langle out \rangle \mapsto s'_i \langle in \rangle$$

is modified to the pair of control dependencies

$$1 : s_i \& s'_i \rightsquigarrow b_i.1 : s_i \& s'_i \rightsquigarrow t'_{i1}$$

Similarly, the following link and data flow relation

$$c < a_1 > : t_{i1} < out > \mapsto t'_{i1} < in > . t'_{i1} < out > \rightarrow v'_1$$

get converted to control dependencies and new data relations as follows

$$1 : t_{i1} \& t'_{i1} \rightsquigarrow t_{i2} . 1 : t_{i1} \& t'_{i1} \rightsquigarrow t'_{i2} . t_{i1} < out > \rightarrow v'_1$$

The transformed model is shown in Figure 4.3(b). Next, we use variable merging rule to merge  $v_1$  and  $v'_1$  (Step 2). We have assumed here that concurrent behaviors may not write to the same variable. Therefore  $v'_1$  has no other writers except  $t_{i1}$  and the condition for applying the variable merging rule is met. Also, we eliminate identity  $t_{i1}$  using Case (a) of the identity elimination rule. As a result, we have the following new control relations added to the model

$$1 : b_i \& t'_{i1} \rightsquigarrow t'_{i2} . 1 : b_i \& t'_{i1} \rightsquigarrow t_{i2}$$

The resulting model is shown in Figure 4.3(c).

In Step 3, we optimize away identity behavior  $t'_{i1}$  using Case (a) of the identity elimination rule. The resulting model, shown in Figure 4.3(d) has the following new control flow relations

$$1 : b_i \& s_i \& s'_i \rightsquigarrow t'_{i2} . 1 : b_i \& s_i \& s'_i \rightsquigarrow t_{i2} \quad (4.1)$$

It can be easily seen that  $s_i \triangleright b_i$  and  $s'_i \triangleright b_i$ . Therefore, we can apply the RCDE rule to remove the redundant control flow relations. Specifically, both  $s_i$  and  $s'_i$  are removed as predecessors from both control flow relations in 4.1. Therefore, we have the resulting model as shown in Figure 4.3(e).

We iteratively apply steps 1 through 4 over all the leaf level behaviors in the model, thereby generalizing for  $b_i$ . In the transformed model in 4.3(e), we can see that a relation

$$q : \dots \& b_i \& \dots \rightsquigarrow b_j \in \bar{M}_s$$

gets transformed into

$$1 : b_i \rightsquigarrow t_i \& 1 : b_i \rightsquigarrow t'_i . q : \dots \& t_i \& \dots \rightsquigarrow s_j . q : \dots \& t'_i \& \dots \rightsquigarrow s'_j . 1 : s_j \& s'_j \rightsquigarrow b_j \in \bar{M}_a$$

Applying identity elimination on  $t_i, t'_i, s_j, s'_j$ , and RCDE we get

$$q : \dots \& b_i \& \dots \rightsquigarrow b_j \in \bar{M}_a$$

Therefore, we have

$$\bar{M}_s = \bar{M}_a \Rightarrow M_s = M_a$$

by soundness of flattening transformation rule. Hence Algorithm 2 is functionality preserving.

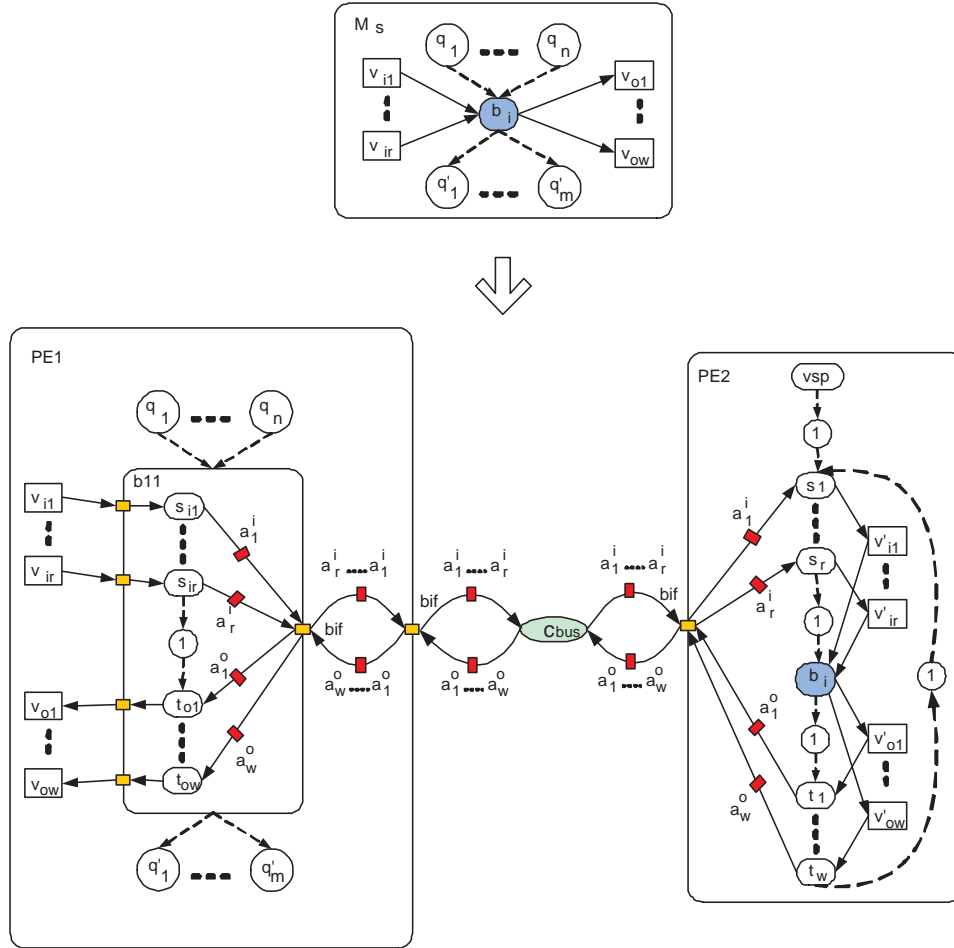


Figure 4.4: RPC style output of behavior partitioning refinement

#### 4.2.4 Refinement Algorithm for RPC style output

The lock-step style output is easy to generate, but may result in too many inter-PE transactions than needed. Some optimizations may be performed to minimize the number of such

transactions while keeping the functional equivalence of the input and output models. In the special case, when a single behavior is mapped to a different PE, one may generate an optimal style of architecture model that has very little inter-PE traffic. Essentially, the behavior mapped to the different PE is triggered into execution as if it were a remote procedure call.

Figure 4.4 shows the output model generated when leaf level behavior  $b_i$  in the specification is mapped to PE2, when the rest of the specification is mapped to PE1. Therefore the partitioning decision is such that

$$map_1 = Leafs(M_s) - \{b_i\}; map_2 = b_i$$

In order to generate the architecture model  $M_a$ , we create a parallel composition of the two PEs as follows

$$M_a = [PE_1].[PE_2].1 : vsp \rightsquigarrow PE_1.1 : vsp \rightsquigarrow PE_2.1 : PE_1 \& PE_2 \rightsquigarrow vtp$$

The specification is copied inside PE1, as is, replacing only  $b_i$  with special hierarchical behavior  $b_{i1}$ . The behavior  $b_{i1}$  is constructed as follows. Let  $\bar{M}_s$  be the specification model that has been flattened down to its leaf level behaviors. Assume that  $b_i$  in  $M_s$  has  $r$  input ports mapped to variables  $v_{i1}$  through  $v_{ir}$ . Also, let  $b_i$  have  $w$  output ports, mapped to variables  $v_{o1}$  through  $v_{ow}$  inside  $\bar{M}_s$ . We create  $b_{i1}$  to be a sequential composition of  $(r + w)$  identity behaviors, namely  $s_{i1}, \dots, s_{ir}, t_{o1}, \dots, t_{ow}$  in that order. We copy all the ports of  $b_i$  in  $M_s$  in  $b_{i1}$  and create port mappings such that  $k^{th}$  in-port is mapped to in port of identity  $s_{ik}$  inside  $b_{i1}$ . Similarly, the  $k^{th}$  out port is mapped to the out port of identity  $t_{ok}$ . Additionally, we create a bus interface port  $bif$  on  $b_{i1}$ . We create blocking write and read relations inside  $b_{i1}$  as follows

$$\forall k, 1 \leq k \leq r, a_k^i : s_{ik} \langle out \rangle \mapsto I \langle bif \rangle$$

$$\forall k, 1 \leq k \leq w, a_k^o : I \langle bif \rangle \mapsto t_{ok} \langle in \rangle$$

On the other end inside PE2, we create a while loop with the body being a sequential composition of  $r + w + 1$  behaviors, namely  $s_1, \dots, s_r, b_i, t_1, \dots, t_w$ , in that order, where all the behaviors, except  $b_i$  are identity. The loop runs endlessly over this body. We create copies of all input and output variables of  $b_i$  in  $M_s$  inside PE2. The  $k^{th}$  input variables is mapped to the out port of  $s_k$ , while the  $k^{th}$  output variable is mapped to the in port of  $t_k$ . We also create blocking read and

write relations inside PE2 as follows

$$\forall k, 1 \leq k \leq r, a_k^i : I \langle bif \rangle \mapsto s_k \langle in \rangle$$

$$\forall k, 1 \leq k \leq r, a_k^i : r_k \langle out \rangle \mapsto I \langle bif \rangle$$

The RPC style output executes as follows. Whenever  $b_i$  is not executing, PE2 is waiting on behavior  $s_1$  for a transaction from PE1. Inside PE1, whenever a control condition triggers execution of  $b_{i1}$ , the sender identity behaviors inside  $b_{i1}$  send the input variables to PE2. Thus, the local copies of the inputs of  $b_i$  are updated whenever  $b_i$  is scheduled to be executed. Once the input gathering transactions are completed,  $b_i$  executes and writes the outputs to PE2's local copy of  $b_i$ 's outputs. Then the sender identity behaviors  $t_1$  through  $t_w$  send the outputs to PE1, where identities  $t_{o1}$  through  $t_{ow}$  update the local copies. The state of PE2 is back to waiting for the next "call". The execution seems to PE1 like a remote procedure call of  $b_i$ , hence the name for the model. The algorithm can be generalized to any number of PEs as long as PEs other than PE1 have only one behavior mapped to them. Algorithm 3 presents the pseudo code for refinement of specification to an RPC style architecture model.

#### 4.2.5 Correctness Proof for RPC Style Refinement

We now present the correctness proof for RPC style refinement algorithm. Assume that  $M_s$  is the specification model and  $b_i$  is a leaf level behavior in  $M_s$ . For simplicity, we will consider the refinement for two PEs only. The proof can easily be generalized for more PEs. Let PE1 and PE2 be the two PEs in the system architecture. Let all leaf behaviors of  $M_s$  be mapped to PE1, with the exception of  $b_i$ . Behavior  $b_i$  is mapped to PE2. Also, without loss of generality, we assume that  $b_i$  has one input and one output. Figure 4.5(a) shows the flattened architecture model for the said mapping.

We start by applying link resolution rule to eliminate channel  $c_{bus}$  from the model and introducing new control and data relations as follows (Step 1). The transaction link and data dependency relations

$$c \langle a_0 \rangle : s_i \langle out \rangle \mapsto s'_i \langle in \rangle \text{ . } s'_i \langle out \rangle \mapsto v'_1$$

---

**Algorithm 3** Refine2RPC ( $M_s, map_1, map_2$ )

---

```
1:  $PE_1 = \bar{M}_s = flatten(M_s)$ 
2:  $Leafs(\bar{M}_s) = \{b_1, b_2, \dots\}$ 
3: for  $i = 1$  TO  $|Leafs(\bar{M}_s)|$  do
4:    $inputs_i = \{v \mid \exists p \in \mathcal{P}, v \rightarrow b_i \langle p \rangle \in \bar{M}_s\}$ 
5:    $outputs_i = \{v \mid \exists p \in \mathcal{P}, b_i \langle p \rangle \rightarrow v \in \bar{M}_s\}$ 
6:   if  $b_i \in map_2$  then
7:      $last_1 = vsp_i^1, last_2 = vsp_i^2$ 
8:     for  $j = 1$  TO  $|inputs_i|$  do
9:        $b_i^1 = b_i^1.1 : last_1 \rightsquigarrow s_{ij}^1.I \langle inp_{ij} \rangle \rightarrow s_{ij}^1 \langle in \rangle$ 
10:       $b_i^1 = b_i^1.a_j^{in} : s_{ij}^1 \langle out \rangle \mapsto I \langle bif \rangle$ 
11:       $b_i^2 = b_i^2.1 : last_2 \rightsquigarrow s_{ij}^2.a_j^{in} : I \langle bif \rangle \mapsto s_{ij}^2 \langle in \rangle$ 
12:       $b_i^2 = b_i^2.s_{ij}^2 \langle out \rangle \rightarrow inv_{ij}^2.inv_{ij}^2 \rightarrow b_i \langle inp_{ij} \rangle$ 
13:       $last_1 = s_{ij}^1, last_2 = s_{ij}^2$ 
14:     end for
15:      $b_i^2 = b_i^2.1 : last_2 \rightsquigarrow b_i; last_2 = b_i$ 
16:     for  $k = 1$  TO  $|outputs_i|$  do
17:        $b_i^1 = b_i^1.1 : last_1 \rightsquigarrow t_{ik}^1.a_k^{out} : I \langle bif \rangle \mapsto t_{ik}^1 \langle in \rangle$ 
18:        $b_i^1 = b_i^1.t_{ik}^1 \langle out \rangle \rightarrow I \langle out_{p_{ik}} \rangle$ 
19:        $b_i^2 = b_i^2.1 : last_2 \rightsquigarrow t_{ik}^2.b_i \langle out_{p_{ik}} \rangle \rightarrow outv_{ik}^2.outv_{ik}^2 \rightarrow t_{ik}^2 \langle in \rangle$ 
20:        $b_i^2 = b_i^2.a_k^{out} : t_{ik}^2 \langle out \rangle \mapsto I \langle bif \rangle$ 
21:        $last_1 = t_{ik}^1, last_2 = t_{ik}^2$ 
22:     end for
23:      $b_i^1 = b_i^1.1 : last_1 \rightsquigarrow vtp_i^1; b_i^2 = b_i^2.1 : last_2 \rightsquigarrow s_{i1}^2$ 
24:      $PE_1 = PE_1|_{b_i=b_i}; PE_2 = PE_2.[b_i^2].1 : vsp_2 \rightsquigarrow b_i^2$ 
25:   end if
26: end for
27:  $M_a = [PE_1].[PE_2].1 : vsp \rightsquigarrow PE_1.1 : vsp \rightsquigarrow PE_2.PE_1 \& PE_2 \rightsquigarrow vtp$ 
```

---

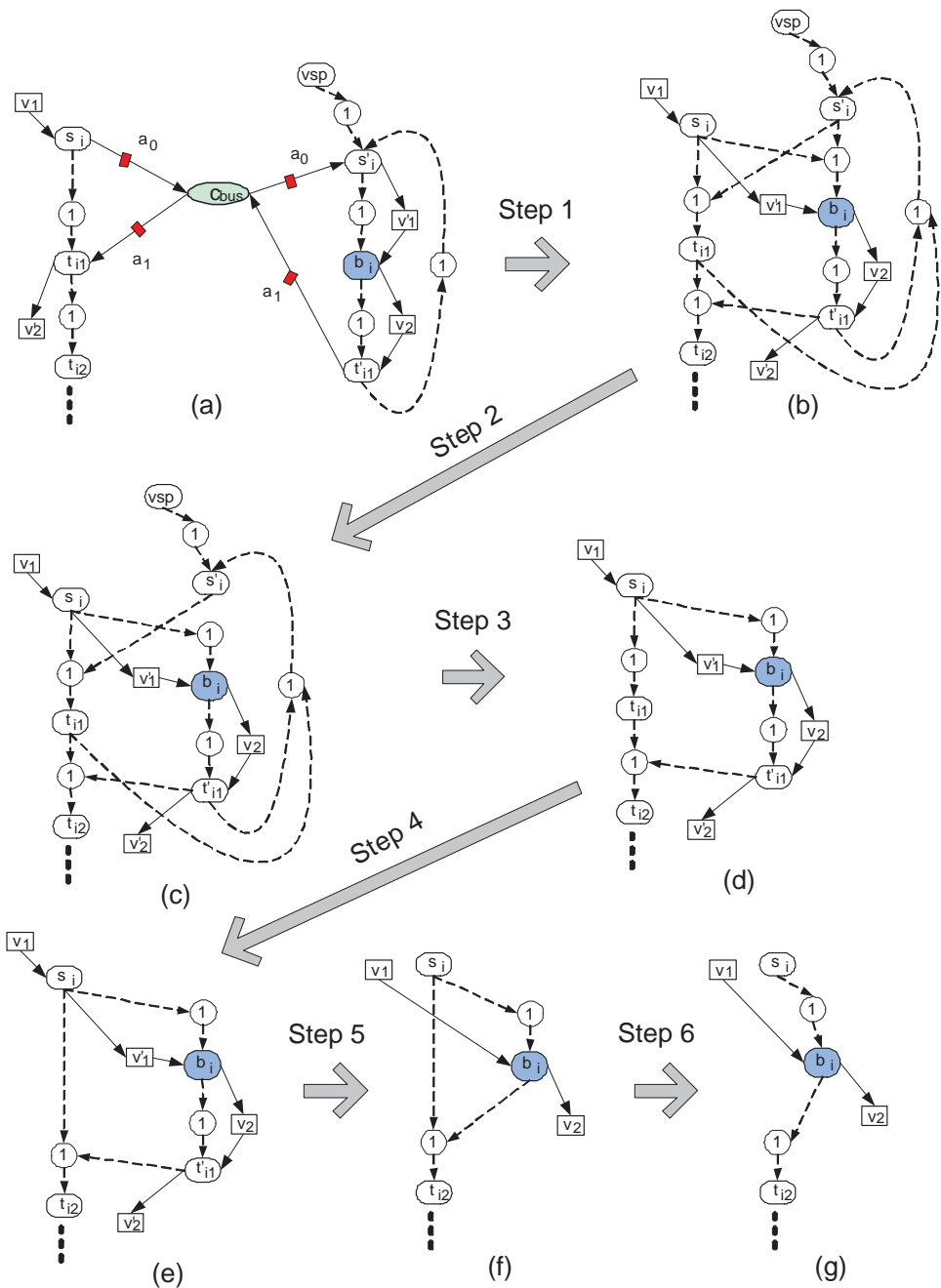


Figure 4.5: Proof steps for refinement to RPC style model

are replaced by new control and data dependency relations

$$1 : s_i \& s'_i \rightsquigarrow t_{i1}. 1 : s_i \& s'_i \rightsquigarrow b_i. 1 : s_i < out > \rightarrow v'_1$$

Also, the following relations

$$c < a_1 > : t'_{i1} < out > \mapsto t_{i1} < in > . t_{i1} < out > \rightarrow v'_2. 1 : t'_{i1} \rightsquigarrow s'_i$$

are replaced by

$$1 : t_{i1} \& t'_{i1} \rightsquigarrow t_{i2}. 1 : t_{i1} \& t'_{i1} \rightsquigarrow s'_i. t_{i1} < out > \rightarrow v'_2$$

The resulting model is shown in figure 4.5(b).

Next, in Step 2, we use a generalization of streamlining transformation rule to remove predecessor  $s'_i$  from the relation

$$1 : s_i \& s'_i \rightsquigarrow b_i \tag{4.2}$$

The streamlining rule is applicable in this case since  $s'_i$  is identity and has two incoming control relations. One relation has  $vsp$  as the predecessor. The other has  $t_{i1}$  and  $t'_{i1}$  as the predecessors. Looking at the control paths from  $s'_i$  to  $t_{i1}$  and  $t'_{i1}$ , it is easy to see that

$$b_i \triangleright t_{i1} \wedge t_{i1} \triangleright (b_i, b_i) \text{ and}$$

$$b_i \triangleright t'_{i1} \wedge t'_{i1} \triangleright (b_i, b_i)$$

Therefore the streamlining rule may be applied to replace the relation in 4.2 by

$$1 : s_i \rightsquigarrow b_i$$

In Step 3, we again apply the streamlining rule, this time replacing

$$1 : s_i \& s'_i \rightsquigarrow t_{i1}$$

with

$$1 : s_i \rightsquigarrow t_{i1}$$

Since  $s'_i$  does not have any out-going edges and it is an identity behavior, it may be removed since it does not modify any variables during model execution. Removal of  $s'_i$  leads to the removal of control conditions leading to  $s'_i$ , thereby resulting in the cleaned up model in Figure 4.5(c).

Using identity elimination rule, we optimize away identity behaviors  $t_{i1}$  and  $t'_{i1}$  in steps 4 and 5, respectively. Before, optimizing away  $t'_{i1}$ , we merge variables  $v_2$  and  $v'_2$  using the variable merging rule. Recall that we do not allow concurrent behaviors to write to the same variable. Therefore, the variable merging rule is applicable in this case. Finally, since  $s_i$  dominates  $b_i$ , we can perform RCDE transformation as shown in Step 6 to derive the model shown in Figure 4.5(g).

Now, by definition of the RPC style refinement algorithm, if there exists a relation

$$q : \dots \& b_j \& \dots \rightsquigarrow b_i \in \bar{M}_s$$

then in the architecture model, we will have

$$q : \dots \& b_j \& \dots \rightsquigarrow s_i \in \bar{M}_a$$

Similarly, if we have

$$q' : \dots \& b_i \& \dots \rightsquigarrow b_k \in \bar{M}_s$$

then in the architecture model, we will have

$$q' : \dots \& t_i \& \dots \rightsquigarrow b_k \in \bar{M}_a$$

Now, from the result in Figure 4.5(g) and optimizing away  $s_i$  and  $t_i$  using identity elimination, we have

$$q : \dots \& b_j \& \dots \rightsquigarrow b_i \in \bar{M}_a \wedge q : \dots \& b_i \& \dots \rightsquigarrow b_k \in \bar{M}_a$$

Therefore, we have

$$\bar{M}_s = \bar{M}_a \Rightarrow M_s = M_a$$

by soundness of flattening transformation rule. Hence, we have proved that behavior partitioning refinement for RPC style output if functionality preserving.

### 4.3 Serializing

After mapping the different tasks in the specification to a set of processing elements, we created a new model that reflected such a mapping. However, this new model might still not be executable on the chosen platform. If we make the assumption that each PE in the system is

a uni-processor then we cannot handle the explicit task level parallelism in the model. In other words, each PE can only have a single controller, therefore if we have any concurrently specified tasks that are mapped to the same PE, then they need to be serialized. Alternately, the designer may include an RTOS that serializes the tasks at run time.

In the context of Model Algebra, concurrent tasks are modeled as being explicitly parallel. If an underlying RTOS does run time thread creation and serialization [YGG03], then it must be verified that the RTOS is indeed emulating the parallel specification as per our trace based notion of equivalence. This requires property checking methods, which are beyond the scope of this dissertation. In this section, we will discuss the refinement that statically serializes parallel behaviors.

### 4.3.1 Design Decisions

The design decision of mapping concurrent tasks to the same PE with static scheduling implies the decision to serialize these tasks. Consider an unscheduled HW PE with two threads of execution. The first thread executes behavior  $b_1$  followed by  $b_2$ , while the second thread executes  $b_3$  followed by  $b_4$ . A possible serialization of the PE would sequentially execute the behaviors in the order  $\{b_1, b_3, b_2, b_4\}$ . Other schedules, that do not violate data dependencies, are also possible.

We now formally define the design decision for serializing. Given a hierarchical behavior  $b$  with parallel composition of its sub-behaviors  $b_1, b_2, \dots, b_n$ . Therefore, we have the control relations as

$$b = [b_1].[b_2]...[b_n].1 : vsp \rightsquigarrow b_1...1 : vsp \rightsquigarrow b_n.1 : b_1 \& \dots \& b_n \rightsquigarrow vtp$$

Let there be an ordered set  $O$ , with  $n$  integer variables such that

$$O = \{o_1, o_2, \dots, o_n\}$$

Let there be an assignment  $A$  of the integers in  $O$  such that each  $o_i, 1 \leq i \leq n$  is assigned a unique integer between 1 and  $n$ . The serialization decision may be given as the set  $O$  and a serialization refinement would modify the control condition of  $b$ , such that after refinement

$$b = [b_1].[b_2]...[b_n].1 : vsp \rightsquigarrow b_{o_1}.1 : vsp \rightsquigarrow b_{o_2}...1 : vsp \rightsquigarrow b_{o_n}.1 : b_1 \& \dots \& b_n \rightsquigarrow vtp$$

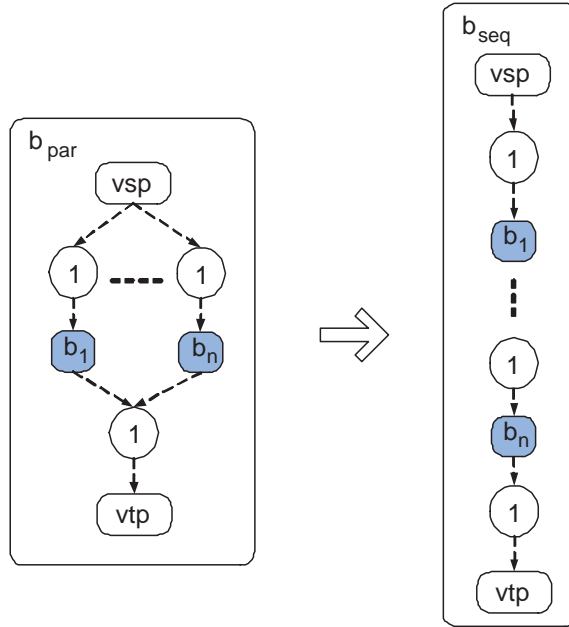


Figure 4.6: Serialization of parallel behaviors

### 4.3.2 Refinement Algorithm

The goal is to generate a model that represents the new ordering of behaviors inside components. Algorithm 4 gives the simple serialization refinement algorithm that serializes a parallel composition according to user decisions. Without loss of generality, we can assume that decision to simply be

$$\forall 1 \leq i \leq n, o_i = i$$

Let there be a parallel composition of behaviors inside a hierarchical behavior  $b_{par}$  in model  $M_s$ .

Let the child behaviors be  $b_1, b_2, \dots, b_n$ . Hence, we have

$$b_{par} = [b_1] \dots [b_n].1 : vsp \rightsquigarrow b_1 \dots 1 : vsp \rightsquigarrow b_n.1 : b_1 \& b_2 \& \dots \& b_n \rightsquigarrow vtp$$

Let  $M_{ser}$  be the model after serialization of  $b_{par}$  inside  $M_s$ .

### 4.3.3 Proof of Correctness

We use induction and control relaxation rules of model algebra to prove the correctness of serialization refinement. The induction step of the proof is shown in Figure 4.7. Assume we

---

**Algorithm 4** Serialize ( $b_{par}, M_s$ )

---

1:  $\bar{M}_s = flatten(M_s)$

**Require:**  $\bar{\Delta}v \in \mathcal{V}, i, j, 1 \leq i, j \leq n, i \neq j$ , s.t.  $b_i < p_i > \rightarrow v \in \bar{M}_s$

**Require:**  $(v \rightarrow b_j < p_j > \in \bar{M}_s \vee b_j < p_j > \rightarrow v \in \bar{M}_s)$

2:  $b_{seq} = b_{par} - 1 : b_1 \& b_2 \& \dots \& b_n \rightsquigarrow vtp$

3:  $last = vsp$

4: **for**  $i = 1$  TO  $n$  **do**

5:    $b_{seq} = (b_{seq} - 1 : vsp \rightsquigarrow b_i)1 : last \rightsquigarrow b_i$

6:    $last = b_i$

7: **end for**

8:  $b_{seq} = b_{seq} \cdot 1 : b_n \rightsquigarrow vtp$

9:  $M_{ser} = M_s |_{b_{par}=b_{seq}}$

---

have a sequential composition of two behaviors  $b_1$  and  $b_2$ , such that there is no data dependency between them. This includes variables that may be written by both behaviors. We have

$$b = [b_1].[b_2].1 : vsp \rightsquigarrow b_1.1 : b_1 \rightsquigarrow b_2.1 : b_2 \rightsquigarrow vtp$$

Since there is no data dependency between  $b_1$  and  $b_2$ , we can use the control relaxation rule to relax the dependency

$$1 : b_1 \rightsquigarrow b_2$$

As a result, the transformed model is a parallel composition as follows

$$b = [b_1].[b_2].1 : vsp \rightsquigarrow b_1.1 : vsp \rightsquigarrow b_2.1 : b_1 \& b_2 \rightsquigarrow vtp$$

Hence, the parallel composition and sequential composition of  $b_1$  and  $b_2$  is equivalent. Therefore, we have shown that serialization algorithm is functionality preserving for two behaviors.

Let us now consider a sequential composition of  $n$  behaviors, named  $b_1$  through  $b_n$ . Assume that using control relaxation, we have *parallelized* the first  $m$  behaviors  $b_1$  through  $b_m$ , where  $m < n$ , as shown in Figure 4.7. We now try to make a parallel composition of  $b_1$  through  $b_{m+1}$ .

Consider control flow relation

$$b_1 \& \dots \& b_m \rightsquigarrow b_{m+1} \tag{4.3}$$

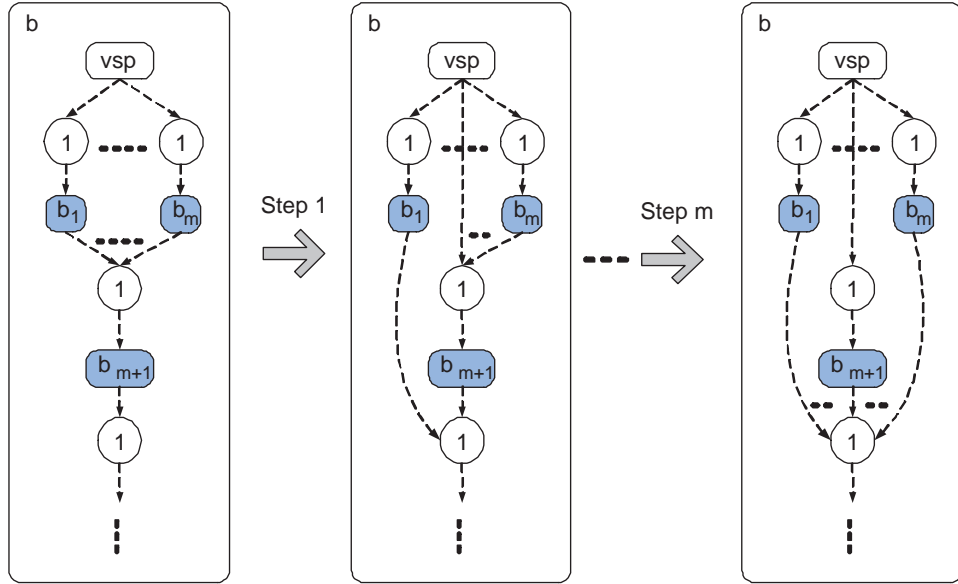


Figure 4.7: Inductive proof steps for serialization of parallel behaviors

Since there is no data dependency between  $b_1$  and  $b_{m+1}$ , the control dependency from  $b_1$  to  $b_{m+1}$  in 4.3 can be relaxed (Step 1). After control relaxation, we get the intermediate model as shown in middle of Figure 4.7. The relation in 4.3 is replaced with two new control relations as follows

$$1 : vsp \& b_2 \& \dots \& b_m \rightsquigarrow b_{m+1} \cdot 1 : b_1 \& b_{m+1} \rightsquigarrow b_{m+2}$$

We continue to relax dependencies from  $b_i$  to  $b_{m+1}$ , where  $1 \leq i \leq m$ . At the  $i^{th}$  step, the control relation in 4.3 is replaced by

$$1 : vsp \& b_{i+1} \& \dots \& b_m \rightsquigarrow b_{m+1} \cdot 1 : b_1 \& \dots \& b_i \& b_{m+1} \rightsquigarrow b_{m+2}$$

This is because  $\forall i, 1 \leq i \leq m, b_i$  does not have any data dependency on  $b_{m+1}$ . Finally, at the  $m^{th}$  step, we get the relations

$$1 : vsp \rightsquigarrow b_{m+1} \cdot 1 : b_1 \& \dots \& b_{m+1} \rightsquigarrow b_{m+2}$$

We have shown that assuming the serialization algorithm is correct for  $m$  behaviors, it will also be correct for  $m + 1$  behaviors. Hence, we have proved in general that the serialization algorithm is functionality preserving.

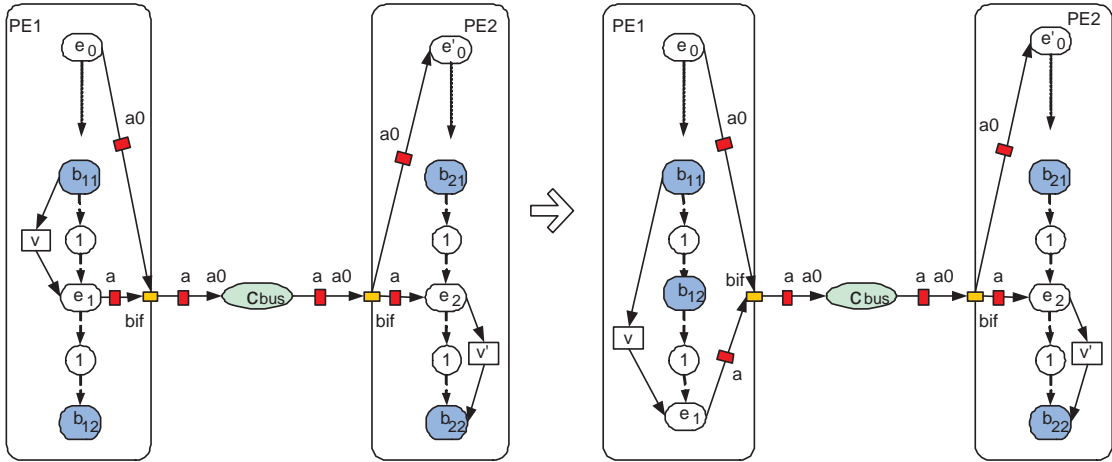


Figure 4.8: Different communication schedules for transaction over channel  $c_{bus}$ .

## 4.4 Communication Scheduling

Once we have partitioned the behaviors and serialized them in their respective PEs, we turn to optimization of inter PE communication. In a system with multiple PEs, point to point communication may be shared over physical busses. Model Algebra, allows the specification of bus based communication, by separating the logical transaction links from physical channels. A single channel may support several logical point to point transaction links. Since channels have built in arbitration, the transactions are ordered randomly at run time. We recall that for functional verification, the actual arbitration scheme is irrelevant as long as it safe and fair. That is, no two transaction requests will be granted together and no transaction request will be forever denied.

One of the important system level optimizations for communication performance is static scheduling of transactions. Specifically, if two transactions are attempted concurrently on a given bus, then it is difficult to predict from the model which transaction will be granted first. Therefore, designer statically schedule transactions on the same bus to improve predictability of performance. Essentially, this results in the reordering of behaviors that have channel transactions. Such a scenario is shown in Figure 4.8, where data  $d$  is sent from PE1 to PE2 over channel  $c$ . The channel implements rendezvous communication semantics, i.e. both sender and receiver must synchronize for the transaction to take place. Consequently, for the case shown in Figure 4.8(a),  $b_{12}$  must wait until  $b_{21}$  has completed and the transaction is performed. If  $b_{21}$  takes a long time to execute, execution inside PE1 will stall, as it waits for the data transaction. Behavior  $b_{12}$  may be scheduled

before the transaction, if it has no data dependency on  $e_1$  or  $e_2$ . Of course, we are assuming that  $b_{12}$  is a leaf level behavior without any blocking relations. The resulting schedule, shown in 4.8(b), optimizes timing so that  $b_{12}$  does not have to wait for the transaction to complete.

#### 4.4.1 Design Decisions

The design decisions for communication scheduling are to advance or postpone a transaction depending on the schedule of bus transaction attempts and traffic [LDR01]. The transaction may be advanced either by reordering the behaviors in the sender PE, such that the sender identity is executed earlier. Alternately, the receiver identity may be scheduled earlier. The exact results of the scheduling depend on run time data, so it cannot be guaranteed statically that a certain reordering would work better. The designer also has a limited window for scheduling the transaction. For instance, the sender identity cannot be scheduled before the data to be sent is available. Conversely, the receiver identity cannot be scheduled after the variable that stores the received data is read. Both these cases will result in erroneous simulation results.

The communication scheduling decisions can essentially be given as a sequence of steps of the following four types

1. Advance sender identity to execute *before* its current scheduled time
2. Postpone sender identity to execute *after* its current scheduled time
3. Advance receiver identity to execute *before* its current scheduled time
4. Postpone receiver identity to execute *after* its current scheduled time

#### 4.4.2 Refinement Algorithm

The goal of communication scheduling refinement is to generate a model that represents the new ordering of behaviors and transactions. We present here four refinement algorithms for the four types of design decisions. Algorithm 5 advances the sender identity before its current schedule by exchanging its position with the immediately preceding behavior in the control flow. Algorithm 6 postpones the execution of sender identity by exchanging its position with the behavior scheduled immediately after the sender in the control flow. Algorithm 7 advances the

receiver identity before its current schedule by exchanging its position with the immediately preceding behavior in the control flow. Algorithm 8 postpones the execution of receiver identity by exchanging its position with the behavior scheduled immediately after the receiver in the control flow. All these refinement methods require that there is no data dependency between the identity and the behavior it exchanges position with. Also, we require that there is a synchronization transaction before either of the exchanged behaviors. Also, the behavior must be leaf level and have no blocking relations.

---

**Algorithm 5** MoveSenderUp (Identity  $e$ , Behavior  $b$ , Behavior  $PE$ )

---

**Require:**  $\exists v \in \mathcal{V}, a \in \mathcal{A}$ , s.t.  $v \rightarrow e \langle in \rangle \in PE, a : e \langle out \rangle \mapsto I \langle bif \rangle \in PE$

**Require:**  $1 : b \rightsquigarrow e \in PE, \nexists b' (\neq b, \neq e) \in \mathcal{B}, q' \in Q$ , s.t.

$$q' : \dots \& b' \& \dots \rightsquigarrow e \in PE \vee q' : \dots \& b \& \dots \rightsquigarrow b'$$

**Require:**  $\nexists a' \in \mathcal{A}, p \in \mathcal{P}$ , s.t.  $a : b \langle p \rangle \mapsto I \langle bif \rangle \in PE \vee$

$$a : I \langle bif \rangle \mapsto b \langle p \rangle \in PE \vee b \langle p \rangle \rightarrow v \in PE$$

1: **for all**  $q \in Q$ , s.t.  $q : b_1 \& \dots \& b_n \rightsquigarrow b \in PE$  **do**

2:  $PE = (PE - q : b_1 \& \dots \& b_n \rightsquigarrow b). q : b_1 \& \dots \& b_n \rightsquigarrow e$

3: **end for**

4: **for all**  $q \in Q, b' \in \mathcal{B}$ , s.t.  $q : \dots \& e \& \dots \rightsquigarrow b' \in PE$  **do**

5:  $PE = (PE - q : \dots \& e \& \dots \rightsquigarrow b'). q : \dots \& b \& \dots \rightsquigarrow b'$

6: **end for**

7:  $PE = (PE - 1 : b \rightsquigarrow e). 1 : e \rightsquigarrow b$

---

### 4.4.3 Proof of Correctness

The proof steps for correctness of communication scheduling refinement are presented in Figure 4.9. In this proof, we only consider the refinement that moves the communicating behavior down on the sender component. The other three cases can be proved similarly.

We start with a flattened model for the one shown in LHS of Figure 4.8. The model has sequentially executing behaviors on both the sender and receiver PE. We see two transactions over the channel  $c_{bus}$ , one at the top between  $e_0$  and  $e'_0$ , labeled  $a_0$  and one between  $e_1$  and  $e_2$ , labeled  $a$ . We are trying to postpone the schedule of the latter transaction. It is assumed that there are

---

**Algorithm 6** MoveSenderDown (*Identitye, Behaviorb, BehaviorPE*)

---

**Require:**  $\exists v \in \mathcal{V}, a \in \mathcal{A}, \text{ s.t. } v \rightarrow e \langle in \rangle \in PE, a : e \langle out \rangle \mapsto I \langle bif \rangle \in PE$

**Require:**  $1 : e \rightsquigarrow b \in PE, \nexists b' (\neq e, \neq b) \in \mathcal{B}, q' \in Q, \text{ s.t.}$

$$q' : \dots \& b' \& \dots \rightsquigarrow b \in PE \vee q' : \dots \& e \& \dots \rightsquigarrow b'$$

**Require:**  $\exists a' \in \mathcal{A}, p \in \mathcal{P}, \text{ s.t. } a : b \langle p \rangle \mapsto I \langle bif \rangle \in PE \vee$

$$a : I \langle bif \rangle \mapsto b \langle p \rangle \in PE \vee b \langle p \rangle \rightarrow v \in PE$$

1: **for all**  $q \in Q, \text{ s.t. } q : b_1 \& \dots \& b_n \rightsquigarrow e \in PE$  **do**

2:  $PE = (PE - q : b_1 \& \dots \& b_n \rightsquigarrow e). q : b_1 \& \dots \& b_n \rightsquigarrow b$

3: **end for**

4: **for all**  $q \in Q, b' \in \mathcal{B}, \text{ s.t. } q : \dots \& b \& \dots \rightsquigarrow b' \in PE$  **do**

5:  $PE = (PE - q : \dots \& b \& \dots \rightsquigarrow b'). q : \dots \& e \& \dots \rightsquigarrow b'$

6: **end for**

7:  $PE = (PE - 1 : e \rightsquigarrow b). 1 : b \rightsquigarrow e$

---

---

**Algorithm 7** MoveReceiverUp (*Identitye, Behaviorb, BehaviorPE*)

---

**Require:**  $\exists v \in \mathcal{V}, a \in \mathcal{A}, \text{ s.t. } e \langle out \rangle \rightarrow v \in PE, a : I \langle bif \rangle \mapsto e \langle in \rangle \in PE$

**Require:**  $1 : b \rightsquigarrow e \in PE, \nexists b' (\neq b, \neq e) \in \mathcal{B}, q' \in Q, \text{ s.t.}$

$$q' : \dots \& b' \& \dots \rightsquigarrow e \in PE \vee q' : \dots \& b \& \dots \rightsquigarrow b'$$

**Require:**  $\exists a' \in \mathcal{A}, p \in \mathcal{P}, \text{ s.t. } a : b \langle p \rangle \mapsto I \langle bif \rangle \in PE \vee$

$$a : I \langle bif \rangle \mapsto b \langle p \rangle \in PE \vee v \rightarrow b \langle p \rangle \in PE$$

1: **for all**  $q \in Q, \text{ s.t. } q : b_1 \& \dots \& b_n \rightsquigarrow b \in PE$  **do**

2:  $PE = (PE - q : b_1 \& \dots \& b_n \rightsquigarrow b). q : b_1 \& \dots \& b_n \rightsquigarrow e$

3: **end for**

4: **for all**  $q \in Q, b' \in \mathcal{B}, \text{ s.t. } q : \dots \& e \& \dots \rightsquigarrow b' \in PE$  **do**

5:  $PE = (PE - q : \dots \& e \& \dots \rightsquigarrow b'). q : \dots \& b \& \dots \rightsquigarrow b'$

6: **end for**

7:  $PE = (PE - 1 : b \rightsquigarrow e). 1 : e \rightsquigarrow b$

---

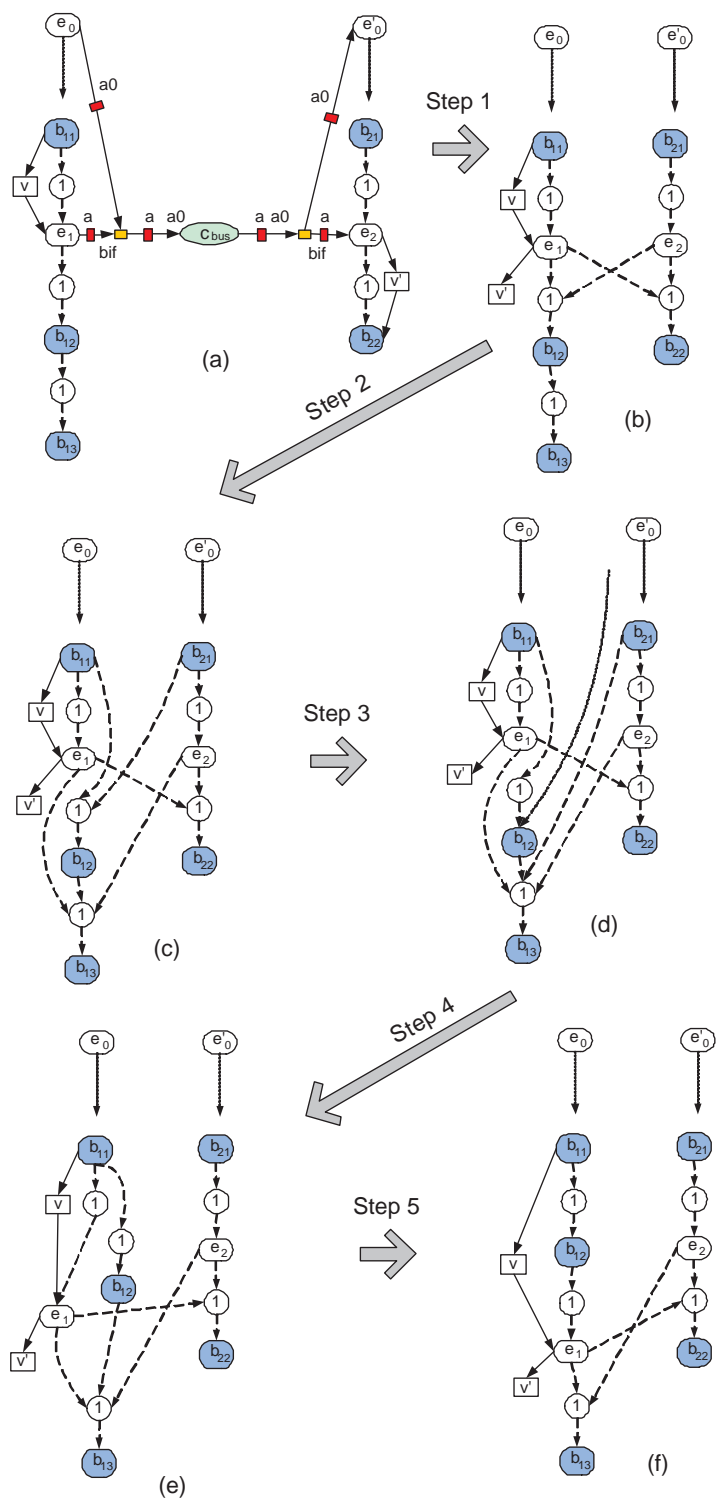


Figure 4.9: Proof steps for verification of communication scheduling

---

**Algorithm 8** MoveReceiverDown (*Identitye, Behaviorb, BehaviorPE*)
 

---

**Require:**  $\exists v \in \mathcal{V}, a \in \mathcal{A}$ , s.t.  $e \langle out \rangle \rightarrow v \in PE, a : I \langle bif \rangle \mapsto e \langle in \rangle \in PE$

**Require:**  $1 : e \rightsquigarrow b \in PE, \nexists b' (\neq e, \neq b) \in \mathcal{B}, q' \in Q$ , s.t.

$$q' : \dots \& b' \& \dots \rightsquigarrow b \in PE \vee q' : \dots \& e \& \dots \rightsquigarrow b'$$

**Require:**  $\nexists a' \in \mathcal{A}, p \in \mathcal{P}$ , s.t.  $a : b \langle p \rangle \mapsto I \langle bif \rangle \in PE \vee$

$$a : I \langle bif \rangle \mapsto b \langle p \rangle \in PE \vee v \rightarrow b \langle p \rangle \in PE$$

1: **for all**  $q \in Q$ , s.t.  $q : b_1 \& \dots \& b_n \rightsquigarrow e \in PE$  **do**

2:  $PE = (PE - q : b_1 \& \dots \& b_n \rightsquigarrow e). q : b_1 \& \dots \& b_n \rightsquigarrow b$

3: **end for**

4: **for all**  $q \in Q, b' \in \mathcal{B}$ , s.t.  $q : \dots \& b \& \dots \rightsquigarrow b' \in PE$  **do**

5:  $PE = (PE - q : \dots \& b \& \dots \rightsquigarrow b'). q : \dots \& e \& \dots \rightsquigarrow b'$

6: **end for**

7:  $PE = (PE - 1 : e \rightsquigarrow b). 1 : b \rightsquigarrow e$

---

sequential paths from  $e_0$  and  $e'_0$  to  $b_{11}$  and  $b_{21}$  respectively. Therefore, we have

$$(e_0 \triangleright b_{11}) \wedge (e'_0 \triangleright b_{21}) \wedge (e_0 \triangleright b_{21}) \wedge (e'_0 \triangleright b_{11}) \quad (4.4)$$

After applying link resolution rule (Step1), the transaction and data relation

$$c_{bus} \langle a \rangle : e_1 \langle out \rangle \mapsto e_2 \langle in \rangle . e_2 \langle out \rangle \rightarrow v'$$

is converted to appropriate control and data relations

$$1 : e_1 \& e_2 \rightsquigarrow b_{12}. 1 : e_1 \& e_2 \rightsquigarrow b_{22}. e_1 \langle out \rangle \rightarrow v'$$

It can be seen from the model that there are no data dependencies from either  $e_1$  or  $e_2$  to  $b_{12}$ . Therefore, in Step 2, we relax these control dependencies using the control relaxation rule. The resulting model is shown in Figure 4.9(c). Now, we have a control dependency from  $b_{21}$  to  $b_{12}$ . Assuming that there are no shared memories between the two components, then we can assume that there is no data dependency between  $b_{21}$  and  $b_{12}$ . Thus, the control dependency can be relaxed. After relaxation, there will be a control dependency on  $b_{12}$  from a behavior that executes before  $b_{21}$ . We continue to relax control dependencies on  $b_{12}$  until we get a dependency from  $e'_0$ . This is guaranteed to happen since there is a sequential composition in the receiver component.

In step 4, we use the result in 4.4 to remove the redundant control dependency on  $b_{12}$  from  $e'_0$ . The application of RCDE is possible since we have the control dependency as

$$1 : b_{11} \& e'_0 \rightsquigarrow b_{12}$$

Also, we can use RCDE on the control condition

$$1 : b_{21} \& e_2 \rightsquigarrow b_{13}$$

because clearly  $b_{21} \triangleright e_2$ . The resulting model is shown in 4.9(e). We can see that the sequential relation between  $e_1$  and  $b_{12}$  is now transformed to a parallel relation. Since these two do not have any data dependencies, we can use the inverse of the control relaxation rule to derive the model shown in Figure 4.9(f). This is isomorphic to the model that we get by flattening and resolving the transaction links of model shown in RHS of Figure 4.8. Therefore, we have shown that communication scheduling refinement is functionality preserving.

## 4.5 Transaction Routing

After behavior partitioning and scheduling, the system model consists of concurrent behaviors communicating with several channels. Although, the model shows the computation structure correctly, the communication structure still needs to be implemented. In a bus-based SoC communication scheme, the various PEs are connected to system busses. The communication model can thus be represented using channels for busses. All transaction links in the input model are shared over the new *bus channels*.

### 4.5.1 Design Decisions

The design decision in this case is choosing the number of bus channels and mapping the transaction links to the bus channels. In some cases, a transaction link may need to be implemented on several busses. This will require the addition of new identity behaviors to act as transducers, that will allow the routing of transactions over busses.

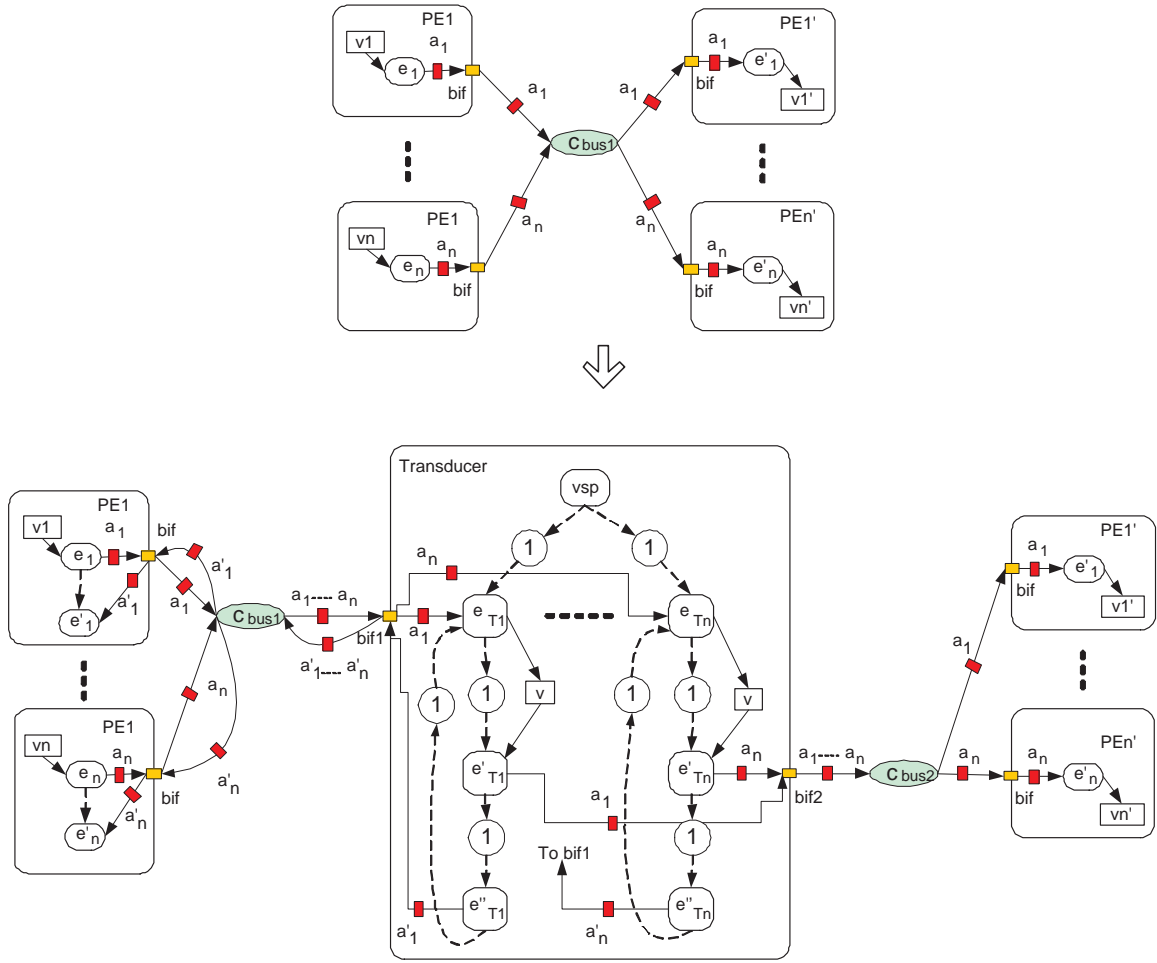


Figure 4.10: Routing transactions via transducer

#### 4.5.2 Refinement Algorithm

Let there be a model  $M_s$  of a system with a bus represented by channel  $c$ . Let there be  $n$  transactions over  $c$  from and to different PEs connected to  $c$ . Suppose we change the communication architecture such that the communicating PEs are now connected to two different busses, represented by channels  $c_1$  and  $c_2$ . The original transaction links represented by addresses  $a_1$  through  $a_n$  must now go over two busses  $c_1$  and  $c_2$ , via a new component called the *Transducer*. The transducer has two interfaces  $bif_1$  and  $bif_2$  for the two busses  $c_1$  and  $c_2$ , respectively. Inside the transducer, we have a parallel composition of  $n$  sets of 3 identity behaviors, each set responsible for routing of transactions on one original link. A set consists of a receiver identity behavior that copies the incoming data from one bus into a local buffer, and a sender identity behavior that

sends the data from buffer over the other bus to the intended recipient and a notification identity behavior that sends a notification transaction to the sender after the sender has executed. Algorithm 9 gives pseudo code for the refinement resulting from transducer insertion.

---

**Algorithm 9** InsertTransducer ( $Address a_1, \dots, a_n, Channel c, c_1, c_2, Behavior M_s$ )

---

```

1:  $M_s = M_s.[T].1 : vsp \rightsquigarrow T$ 
2: for  $i = 1$  to  $n$  do
Require:  $\exists PE_s, PE_r \in \mathcal{B}$ , s.t.  $c < a_i > : PE_s < bif > \mapsto PR_r < bif > \in M_s$ 
3: if ( $ConnectedTo(PE_s) == bus1 \wedge ConnectedTo(PE_r) == bus2$ ) then
4:    $M_s = (M_s - c < a_i > : PE_s < bif > \mapsto PR_r < bif >).c_1 < a_i^1 > : PE_s < bif > \mapsto T < bif_1 >$ 
5:    $M_s = M_s.c_2 < a_i^2 > : T < bif_2 > \mapsto PR_r < bif >$ 
6:    $T = T.1 : vsp_T \rightsquigarrow e_{Ti}.1 : e_{Ti} \rightsquigarrow e'_{Ti}.1 : e'_{Ti} \rightsquigarrow e''_{Ti}.1 : e''_{Ti} \rightsquigarrow e_{Ti}$ 
7:    $T = T.e_{Ti} < out > \rightarrow v_i.v_i \rightarrow e'_{Ti} < in >$ 
8:    $T = T.a_i^1 : I < bif_1 > \mapsto e_{Ti} < in > .a_i^2 : e'_{Ti} < out > \mapsto I < bif_2 > .a_i^3 : e''_{Ti} < out > \mapsto$ 
      $I < bif_1 >$ 
9: else if ( $ConnectedTo(PE_s) == bus2 \wedge ConnectedTo(PE_r) == bus1$ ) then
10:   $M_s = (M_s - c < a_i > : PE_s < bif > \mapsto PR_r < bif >).c_2 < a_i^1 > : PE_s < bif > \mapsto T < bif_2 >$ 
11:   $M_s = M_s.c_1 < a_i^2 > : T < bif_1 > \mapsto PR_r < bif >$ 
12:   $T = T.1 : vsp_T \rightsquigarrow e_{Ti}.1 : e_{Ti} \rightsquigarrow e'_{Ti}.1 : e'_{Ti} \rightsquigarrow e_{Ti}$ 
13:   $T = T.e_{Ti} < out > \rightarrow v_i.v_i \rightarrow e'_{Ti} < in >$ 
14:   $T = T.a_i^2 : I < bif_2 > \mapsto e_{Ti} < in > .a_i^1 : e'_{Ti} < out > \mapsto I < bif_1 >$ 
15: end if
16: end for

```

---

### 4.5.3 Proof of Correctness

Figure 4.11 shows the basic proof steps for showing equivalence between models before and after transducer insertion. Without loss of generality, we assume a transducer between two channels  $c_1$  and  $c_2$  as shown in the flattened model in Figure 4.11(a). The transducer replaces a direct channel transaction between identities  $e_1$  and  $e_2$ . The transducer consists of an endless loop with a sequential body of 3 identities namely  $e_T$ ,  $e'_T$  and  $e''_T$ . Identity  $e_T$  has a transaction link from

$e_1$ . After this transaction occurs,  $e_T$  copies the received data in a local variable  $v$ . Thus  $v$  is a copy of  $v_1$ . Then,  $e'_T$  executes, sending data from  $v$  to  $e_2$  over channel  $c_2$ . Finally, after this transaction is complete, identity  $e''_T$  executes and synchronizes with  $e'_1$

We start by resolving all the transaction links in the model into appropriate control and data dependency relations using the link resolution transformation rule (Step 1). By the application of this rule, we replace transaction and data relations

$$c_1 \langle a_1 \rangle : e_1 \langle out \rangle \mapsto e_T \langle in \rangle . e_T \langle out \rangle \rightarrow v$$

with the following relations

$$1 : e_1 \& e_T \rightsquigarrow e'_T . 1 : e_1 \& e_T \rightsquigarrow e'_1 . e_1 \langle out \rangle \rightarrow v$$

Transaction relations between  $e'_T$  and  $e_2$  and data relations of  $e_2$  as follows

$$c_2 \langle a_2 \rangle : e'_T \langle out \rangle \mapsto e_2 \langle in \rangle . e_2 \langle out \rangle \rightarrow v_2$$

are replaced by control and data relations

$$1 : e_2 \& e'_T \rightsquigarrow e''_T . 1 : e_2 \& e'_T \rightsquigarrow e'_2 . e_2 \langle out \rangle \rightarrow v_2$$

The notification transaction between  $e'_1$  and  $e''_T$  is also replaced similarly using the link resolution rule. At the end of transformations of Step 1, the model looks like as shown in Figure 4.11(b).

In Step 2, we apply the streamlining rule in the same fashion as used in the proof for RPC style behavior partitioning algorithm. The rule eliminated control dependencies from  $e_T$  to  $e'_1$  and from  $e_T$  to  $e'_T$ . As a result,  $e_T$  does not have anything executing after it and it does not modify and variable in the model, so it is removed from the model along with all its control and data relation. The resulting model after this transformation is showed in Figure 4.11(c).

In Step 3, we optimize away identities  $e'_T$  and  $e'_1$  using the identity elimination rule. In Step 4, we use the RCDE rule to remove the redundant control dependency from  $e_1$  to  $e''_1$ . This is because the control dependency is cause by the relation

$$1 : e_1 \& e''_T \rightsquigarrow e''_1$$

and  $e_1 \triangleright e''_T$ , there by allowing application of RCDE. Finally, in Step 5, we optimize away identity  $e''_T$  using the identity elimination rule. The final resulting model, as shown in Figure 4.11(f), is

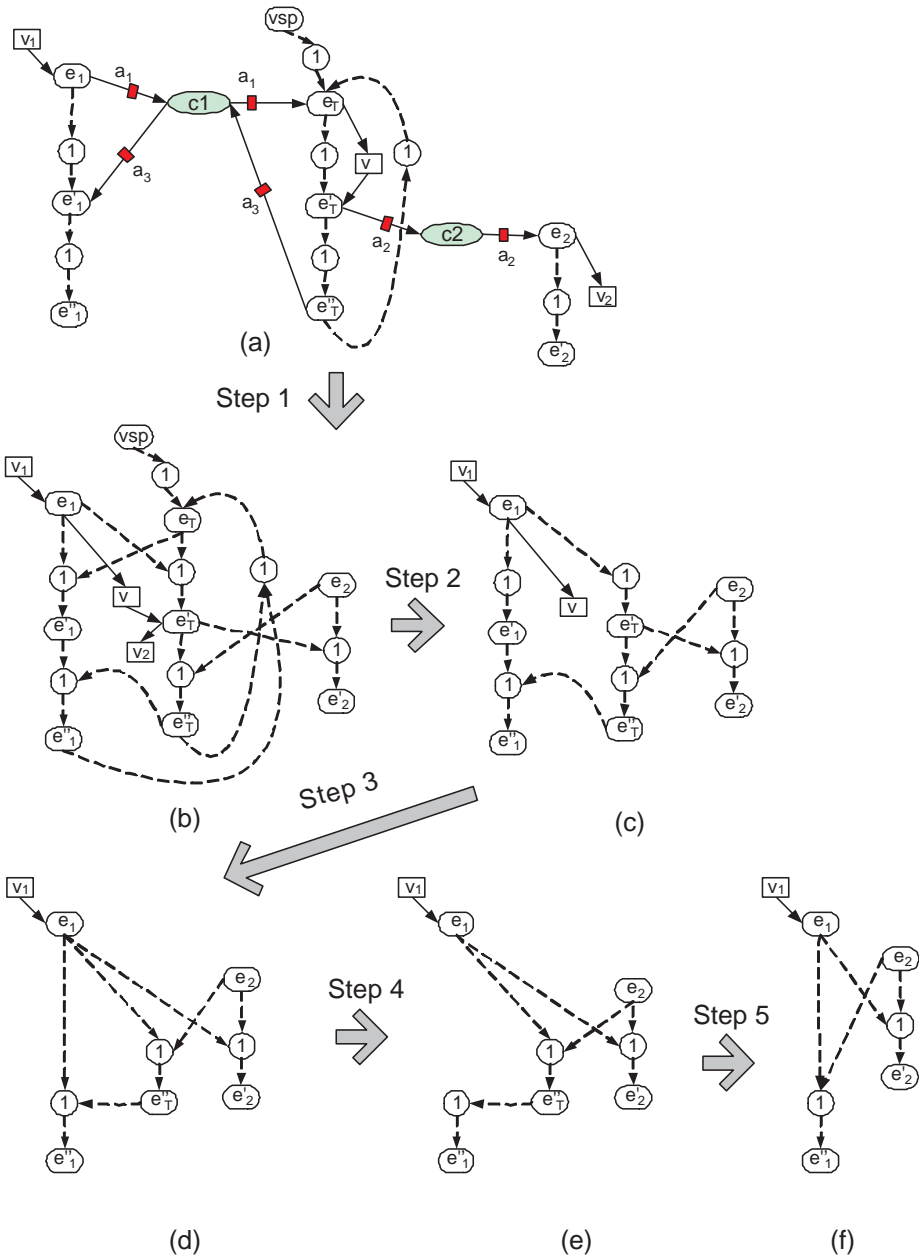


Figure 4.11: Proof steps for transducer insertion during transaction routing refinement

identical to the one derived after link resolution of a direct link from  $e_1$  to  $e_2$ . Therefore, we have proved that the transducer insertion algorithm is functionality preserving.

## 4.6 Chapter Summary

In the last chapter, we had introduced our system level formalism and established functionality preserving transformation rules in that formalism. In this chapter, we built upon that theory to present some useful system level refinements. The refinement algorithms were proven for correctness using the transformation rules of Model Algebra.

We presented the behavior partitioning refinements that come in two flavors. Based on the design decision of mapping the tasks in a functional specification to different processing elements (PEs), the refinement algorithms generated two different types of output models that reflected the mapping. The first output model was based on the lock-step style of synchronization, where the state of each PE is synchronized with other PEs. The second output model employed a remote procedure call style of synchronization, where the master PE used a channel transaction to start execution of a behavior mapped to a slave PE. After execution, the slave PE returns to wait for the next request from the master.

We then presented two commonly used refinements for scheduling tasks in different PEs in the system. The first refinement was used to serialize parallel tasks if the designer wants to minimize the number of controllers. The second refinement used reordering of communicating behaviors to change the overall communication schedule in the system. Communication scheduling is used to minimize the number of competing transactions on the bus, which in turn leads to fewer arbitration delays and more predictable system behavior.

Finally, we presented a refinement resulting from changes to the communication architecture in the system. If two communicating PEs have different interface structure or protocol, then they cannot use the a single bus for communication. The transactions between such PEs must be routed through a special bridge behavior that we refer to as a transducer. We showed how a system model may be refined to insert transducers between PEs with incompatible interfaces.

All the aforementioned refinements were proved correct using the rules of Model Algebra. In our design methodology, such refinements are frequently used to evaluate different system

architectures. Reliable methods for implementing these refinements cut the costs for verifying all refined models that are generated during the system level design process.

## Chapter 5

# Automatic Refinement Verification

In the previous chapter we established theoretical results pertaining to the refinement of system level models in Model Algebra and the verification of such refinements. We used the proof strategies for showing correctness of refinements to develop an automatic tool that applies the sound transformation rules of model algebra on two models to prove their functional equivalence. The models themselves are abstractions of performance models written in system design languages, in our case SpecC.

Our verification methodology is shown in Figure 5.1. It follows from our system level design methodology already discussed in the previous chapter and shown in Figure 4.1. We use our automatic tool to check the equivalence of the models that undergo various refinements in our methodology. In this chapter, we will present our equivalence verification tool, discuss the automatic verification algorithms and provide experimental results for the verification tool on SpecC examples.

The verification tool flow is shown in Figure 5.2. First, we abstract the SLDL model into a graphical MA representation. The MA representation then undergoes a series of transformations until no more transformations can be applied. The final model is said to be normalized. The normalized models are then checked for isomorphism. There are two key requirements for such a verification technique to succeed. The first requirement is the *soundness* of the transformation laws that are used to normalize the model. Soundness means that each transformation must produce an equivalent model, according to the equivalence notion defined above. The second requirement

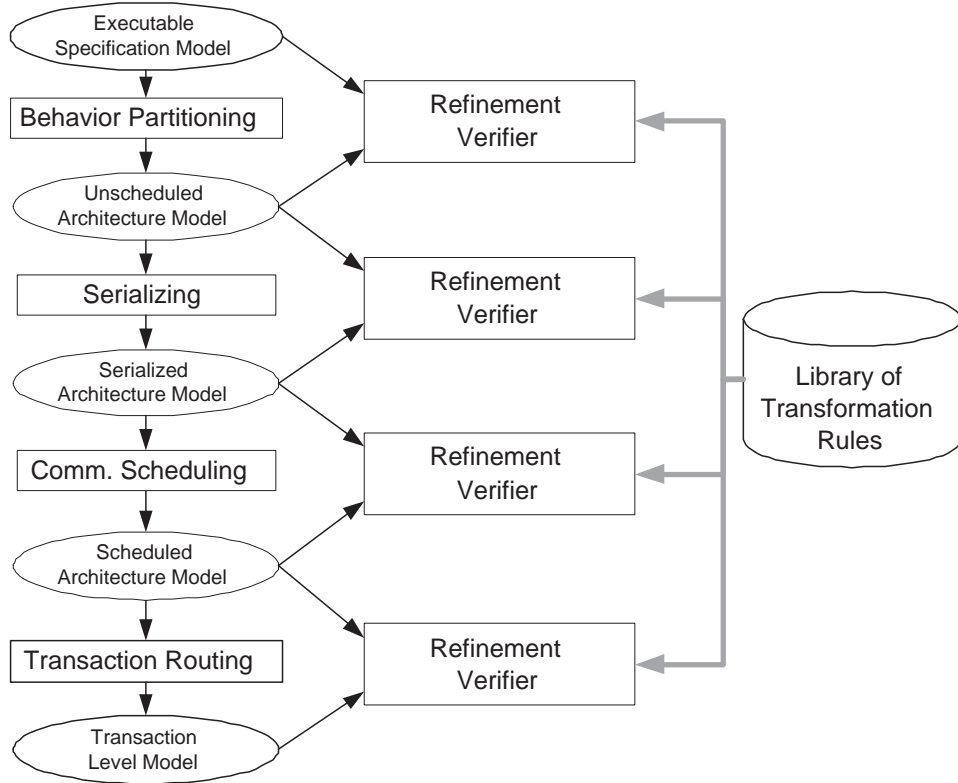


Figure 5.1: Refinement Verification methodology

is that the normalization algorithm must always produce isomorphic normal forms for any two models that can be proven equivalent using the transformation laws.

## 5.1 Functional Abstraction

Our verification tool is based on the syntax and semantics of Model Algebra. However, it would be inconvenient and complicated for verification engineers to convert their system level models into model algebraic expressions for verification. The key difference between a model written in a typical SLDL and once written in Model Algebra is that the model algebraic model treats behaviors as black boxes or uninterpreted functions. Also, it treats any conditional expressions in the model as uninterpreted predicates. The SLDL model evaluates values of variables during execution, while in model algebra, we only care about the interactions and dependencies of the behaviors, conditions and variables. Therefore, a reasonable approach would be to automati-

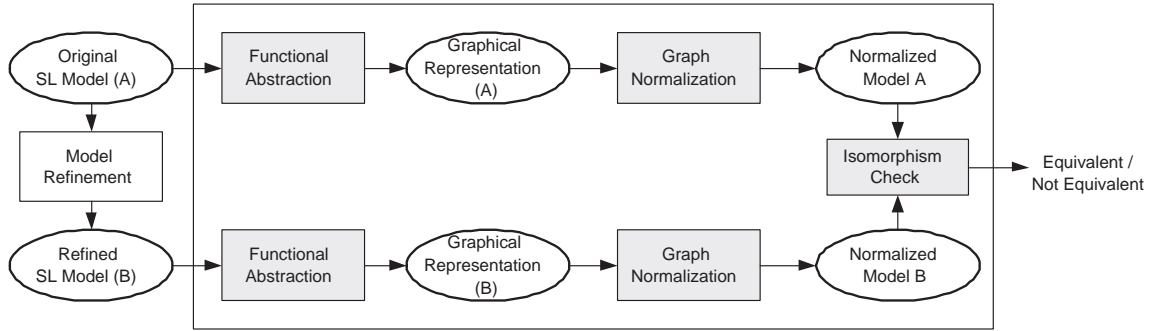


Figure 5.2: Steps in refinement verification

cally derive the model algebraic expression from an SLDL model.

The front end of our tool is essentially a SpecC parser that creates model algebraic expressions from SpecC behaviors. In this section, we will give a brief overview of the relevant features of SpecC and show how a SpecC behavior is abstracted into a model algebraic behavior.

### 5.1.1 SpecC language and SIR

SpecC is a system design language that was created for modeling of embedded systems consisting of tasks that run in HW and those that run in SW. It is built on top of ANSI-C and allows explicit modeling of concurrency, bit-vector types and hierarchy to allow system level representation of HW/SW designs. We now look at the basic structure of SpecC language that are relevant for our model algebra based functional verification. We also discuss the SIR application programming interface that allows us to interpret and manipulate SpecC models. In our verification tool, the key idea of functional abstraction module is to derive the model algebraic expression of a design from its SpecC representation.

#### 5.1.1.1 Basic Structure of SpecC

SpecC allows us to code a hierarchical composition of communicating behaviors. Each behavior is either a module written in ANSI-C or a further composition of other behaviors. We will not consider hierarchical behaviors that have sub-behaviors interspersed with other C-code. It is perfectly legitimate to write and execute such a model, but defeats the purpose of clean hierarchy and treatment of leaf behaviors as uninterpreted functions. SpecC allows for ports and

variables that have identical interpretation as ports and variables in MA. However, in SpecC, ports and variables may have data types, something that is not considered in MA. In general, channels in SpecC are different from those in MA since SpecC syntax considers a channel to be simply a class with available functions. These functions may be declared in an interface that is implemented by that channel. It is possible to have channels with functions for purposes other than communication, which would make them like any other behavior in MA. We will consider only one implementation of the SpecC channel, namely the double handshake channel with mutually exclusive data transfers. The semantics of the channel have already been discussed in previous sections. This channel can be easily written in SpecC and is assumed to be available in the models that we are trying to verify. No other types of channels are allowed.

#### **5.1.1.2 SpecC Internal Representation**

The SpecC internal representation (SIR) is an abstract syntax representation of a model written with SpecC syntax. The SIR also comes with a rich API for reading and modifying SpecC programs. The functional abstraction module uses the SIR API to read a SpecC model and build the verifier's internal data structure representing the hierarchical model algebraic expression of the model. SIR API provides the following classes and types that facilitate creation of model algebraic expressions. *sir\_behavior::BehaviorClass* used to identify if a behavior is to be treated as uninterpreted function or a hierarchical behavior. Control relations are created based on the returned behavior type. Classes for *sir\_portmap* and *sir\_portdirection* used to identify port mappings and the corresponding data flow relations. Inside FSM behaviors *sir\_transitions* are used to create control relations with complex control conditions.

### **5.1.2 Executable Performance Models in SpecC**

A typical executable performance model is shown in Figure 5.3(a). The platform consists of a processor (Proc.) connected to Bus1 along with two hardware components (HW1 and HW2). Communication elements such as arbiter and interrupt controller (IC) allow safe and synchronized communication between the processor and the HW components. The architecture in the model is captured using hierarchy (a parallel composition of components) and signals and

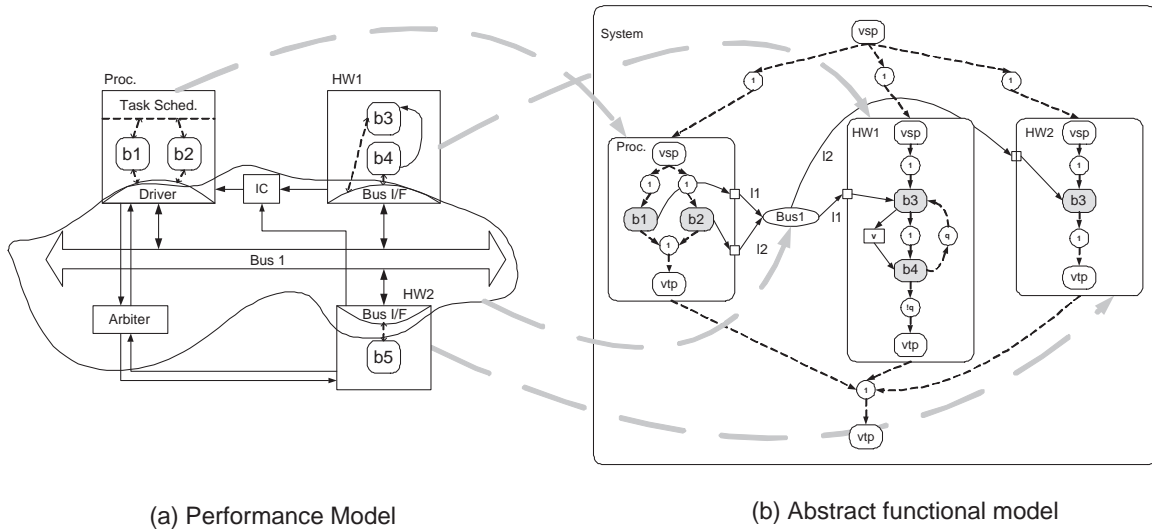


Figure 5.3: The platform model and its corresponding functional abstraction

channels for wires and bus interfaces. The application itself is captured by instantiating behaviors inside the appropriate component behavior. For instance, behaviors  $b_1$  and  $b_2$  of the application are mapped to Proc. as seen by the behavioral hierarchy.

### 5.1.3 Deriving Model Algebraic Expression

Given the syntax of Model Algebraic representation, we derive the abstract functional model of our example as shown in Figure 5.3(b). The bold grey arrows show the abstraction of the objects in the performance model in Figure 5.3(a) into objects in the abstract model on the right hand side. All hierarchical behaviors have unique start and terminate identity behaviors called *virtual starting point* ( $vsp$ ) and *virtual terminating point* ( $vtp$ ) respectively. Leaf behaviors are copied as is since they are treated as uninterpreted functions. Sequentiality and loops inside HW1 are modeled using control dependencies as shown. Note the abstraction of the behaviors and the task scheduler inside Proc. using the control flow of MA. Both behaviors  $b_1$  and  $b_2$  are allowed to start simultaneous after  $vsp$ . However, the Proc. behavior terminates only after both  $b_1$  and  $b_2$  have terminated. All communication implemented in the performance model (using bus signals, arbiter, IC, drivers and bus interfaces) is abstracted into high level transaction links  $l_1$  and  $l_2$  implemented on the channel labeled  $Bus_1$ . Finally, the top level system is abstracted as a parallel composition of hierarchical component behaviors.

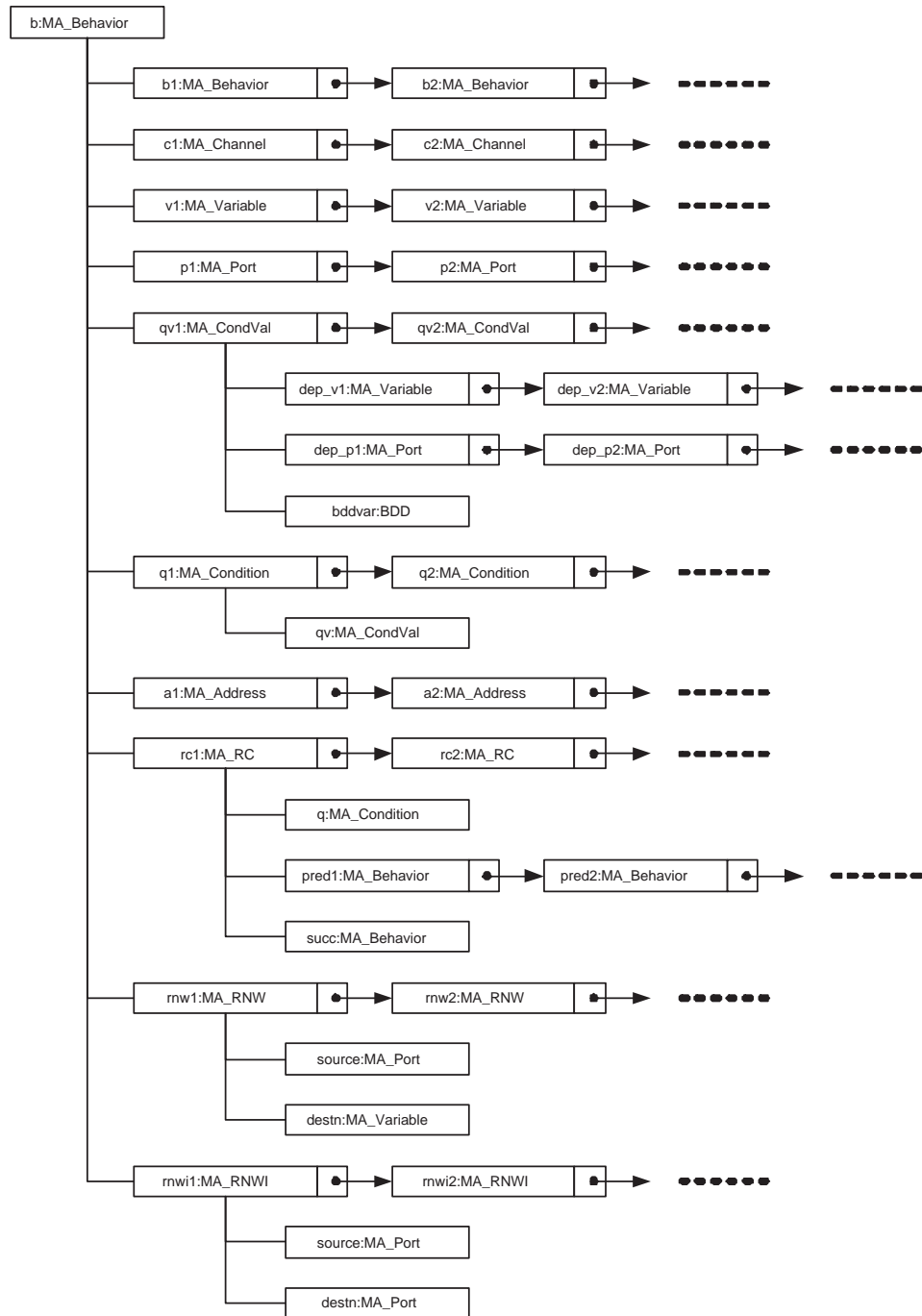


Figure 5.4: Data structure for storing model algebraic expression

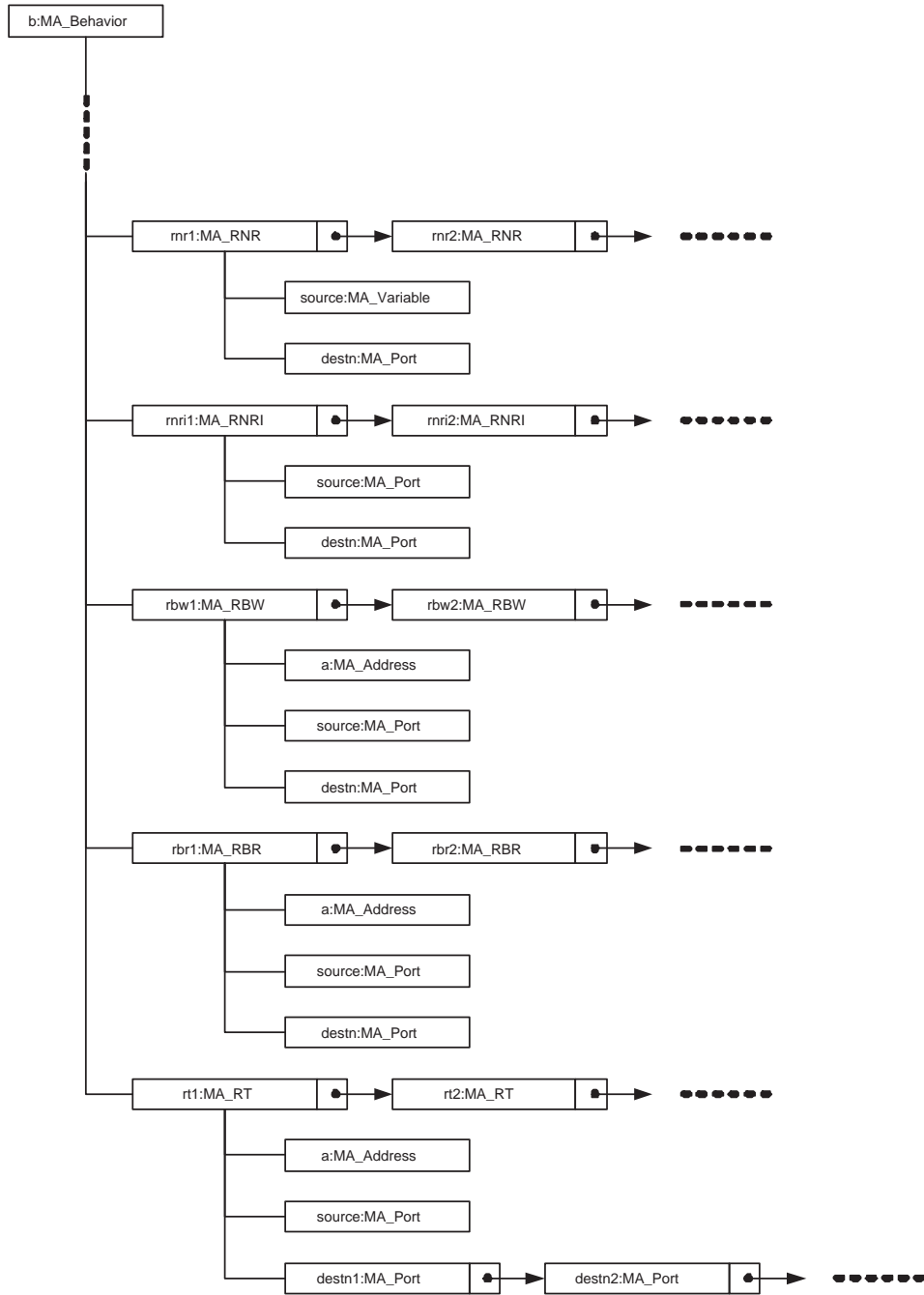


Figure 5.5: Data structure for storing model algebraic expression (contd.)

### 5.1.3.1 Internal representation for MA

In MA, the model is simply the top level behavior. In Figures 5.4 and 5.5, we have shown the data structure for organizing model algebraic expression of a behavior. Each object and composition rule in MA (excluding interface) has a unique class implementation. Additionally, we have a class for boolean functions that are used for representing the values inside control conditions. These boolean functions are stored and manipulated using BDDs in the CUDD package and its application programming interface [CUD].

### 5.1.3.2 Translation from SpecC to MA

The system model in SpecC is a hierarchical composition of behaviors using different composition styles such as sequential, parallel, FSM and pipeline. The behaviors at the same scope communicate with variables or channels. Across hierarchies, ports facilitate behavior communication. There is a direct translation possible from a subset of SpecC to MA. We will assume that for synthesis and refinement purposes, the designers uses only those constructs that are allowed in MA. All other SpecC constructs such as events (outside channels), interrupts and exceptions will be disregarded for our discussion.

The pseudo code in the recursive Algorithm 10 is used to read in the SIR data structure of a hierarchical behavior and convert it to a set of corresponding model algebraic expressions. The algorithm recursively traverses the SpecC behaviors in a depth first fashion and creates terms for behaviors at the current scope. We also give pseudo code for various helper functions that are used during functional abstraction. These functions include Algorithm 11 that is used to create blocking data flow relations from the port mappings to interfaces in SpecC. The method described in Algorithm 12 is used to create transaction link relations from SpecC behavior instantiations and port mapping to channels. The final helper method described in Algorithm 13 is used to create control flow relations from different types of SpecC hierarchical behaviors. In this pseudo code, we will only show the control flow creating from parallel compositions (*par* behavior in SpecC) and FSM behaviors. Other type of supported SpecC behaviors like sequential and pipelined may be derived from FSM and *par* behaviors.

---

**Algorithm 10** SpecC2MA (BehaviorInstance  $scb$ ) returns MA\_Behavior type

---

```
1:  $scbtype = \text{SpecC behavior whose instance } scb \text{ is}$ 
2:  $b \rightarrow \text{PortList} = scbtype \rightarrow \text{PortVariables}$ 
3: if  $scbtype$  is leaf behavior then
4:   RETURN  $b$ 
5: end if
6:  $b \rightarrow \text{VariableList} = scbtype \rightarrow \text{Variables}$ 
7:  $b \rightarrow \text{ChannelList} = scbtype \rightarrow \text{Channels}$ 
8: for all  $scsub \in scbtype \rightarrow \text{SubBehaviorInstances}$  do
9:    $b = b.[\text{SpecC2MA}(scsub)]$ 
10: end for
11: for all  $pmap \in scb \rightarrow \text{PortMaps}$  do
12:   if  $pmap$  maps in-port  $p_{sub}$  of  $scsub$  to  $v \in scbtype \rightarrow \text{Variables}$  then
13:      $b = b.v \rightarrow scsub \langle p_{sub} \rangle$ 
14:   end if
15:   if  $pmap$  maps out-port  $p_{sub}$  of  $scsub$  to  $v \in scbtype \rightarrow \text{Variables}$  then
16:      $b = b.scsub \langle p_{sub} \rangle \rightarrow v$ 
17:   end if
18:   if  $pmap$  maps in-port  $p_{sub}$  of  $scsub$  to  $p \in scbtype \rightarrow \text{PortVariables}$  and  $p.Type ==$ 
     $VARIABLE$  then
19:      $b = b.I \langle p \rangle \rightarrow scsub \langle p_{sub} \rangle$ 
20:   end if
21:   if  $pmap$  maps out-port  $p_{sub}$  of  $scsub$  to  $p \in scbtype \rightarrow \text{PortVariables}$  and  $p.Type ==$ 
     $VARIABLE$  then
22:      $b = b.scsub \langle p_{sub} \rangle \rightarrow I \langle p \rangle$ 
23:   end if
24: end for
25: CreateBlockingDataFlow ( $b, scb$ )
26: CreateControlFlow ( $b, scbtype$ )
```

---

---

**Algorithm 11** CreateBlockingDataFlow (MA\_Behavior  $b$ , BehaviorInstance  $scb$ )

---

```
1: for all  $pmap \in scb \rightarrow PortMaps$  do
2:   if  $pmap$  maps in-port  $p_{sub}$  of  $scsub$  to  $p \in scbtype- > PortVariables$  and  $p.Type ==$ 
   INTERFACE then
3:     if  $scsub$  is Identity and  $p_{sub}.recv(a, ..) \in scsub$  then
4:        $p.addresses \rightarrow Append(a)$ 
5:     else
6:        $p.addresses \rightarrow Append(p_{sub}.addresses)$ 
7:     end if
8:      $b = b.a : I < p > \mapsto scsub < p_{sub} >$ 
9:   end if
10:  if  $pmap$  maps out-port  $p_{sub}$  of  $scsub$  to  $p \in scbtype- > PortVariables$  and  $p.Type ==$ 
   INTERFACE then
11:    if  $scsub$  is Identity and  $p_{sub}.send(a, ..) \in scsub$  then
12:       $p.addresses \rightarrow Append(a)$ 
13:    else
14:       $p.addresses \rightarrow Append(p_{sub}.addresses)$ 
15:    end if
16:     $b = b.a : scsub < p_{sub} > \mapsto I < p >$ 
17:  end if
18: end for
19: CreateTransactions ( $b, scb$ )
```

---

---

**Algorithm 12** CreateTransactions (MA\_Behavior  $b$ , BehaviorInstance  $scb$ )

---

```
1: for all  $pmap \in scb \rightarrow PortMaps$  do
2:   if  $pmap$  maps in-port  $p_{sub}$  of  $scsub$  to  $c \in scbtype \rightarrow Channels$  then
3:     for all  $a \in p_{sub}.addresses$  do
4:        $b.links \rightarrow Append(a)$ 
5:        $a.channel = c$ 
6:        $a.receivers = a.receivers \& scsub \langle p_{sub} \rangle$ 
7:     end for
8:   end if
9:   if  $pmap$  maps out-port  $p_{sub}$  of  $scsub$  to  $c \in scbtype \rightarrow Channels$  then
10:    for all  $a \in p_{sub}.addresses$  do
11:       $b.links \rightarrow Append(a)$ 
12:       $a.channel = c$ 
13:       $a.sender = scsub \langle p_{sub} \rangle$ 
14:    end for
15:   end if
16:   for all  $a \in b.links$  do
17:      $b = b.a.channel \langle a \rangle: a.sender \mapsto a.receivers$ 
18:   end for
19: end for
```

---

---

**Algorithm 13** CreateControlFlow (MA\_Behavior  $b$ , Behavior  $scbtype$ )

---

```
1: if  $scbtype \rightarrow Type == FSM$  then
2:    $b = b.1 : vsp_b \rightsquigarrow scbtype \rightarrow FirstState$ 
3:    $tx = scbtype \rightarrow Transitions \rightarrow First$ 
4:   while  $tx \neq NULL$  do
5:     if  $tx \rightarrow NextState == NULL$  then
6:        $b = b.tx \rightarrow Condition : tx \rightarrow CurrentState \rightsquigarrow vt p_b$ 
7:     else
8:        $b = b.tx \rightarrow Condition : tx \rightarrow CurrentState \rightsquigarrow tx \rightarrow NextState$ 
9:     end if
10:     $tx = tx \rightarrow Next$ 
11:  end while
12: else if  $scbtype \rightarrow Type == PAR$  then
13:    $sub_b = b \rightarrow SubBehaviors \rightarrow First$ 
14:    $term = 1 : sub_b$ 
15:  while  $sub_b \neq NULL$  do
16:     $b = b.1 : vsp_b \rightsquigarrow sub_b$ 
17:    if  $sub_b \neq b \rightarrow SubBehaviors \rightarrow First$  then
18:       $term = term \& sub_b$ 
19:    end if
20:     $sub_b = sub_b \rightarrow Next$ 
21:  end while
22:   $b = b.term \rightsquigarrow vt p_b$ 
23: end if
```

---

## 5.2 Model Normalization

The abstracted MA expression from the input SpecC models must now be reduced to a normal form using the correct transformations in MA. Checking equivalence of two arbitrary MA expressions has not been shown to be decidable. This is because we do not have any results on the completeness of our transformation rules. If we consider two MA expressions such that we can prove their equivalence using our transformation rules, we need to ask if there exists an algorithm, with some reasonable upper bound, that can automatically perform a proof. Clearly, an algorithm that tries all possible transformations will be able to reduce any model to any other provably equivalent model. However, such an algorithm would be intractable and have no practical significance. Hence, we must resort to manually guided proof engines for showing equivalence of models, just like in theorem proving in higher order logics.

Although it seems that MA suffers from the same verification problems as other formalisms, it must be noted that our goal is to verify refinements as opposed to equivalence of arbitrary models. The normalization process mimics the manual proof steps we showed in the last chapter for verification of behaviors partitioning, scheduling and transaction routing. If the refinement algorithms were correctly implemented, the normalization process would reduce the MA expressions of the original and refined SpecC models to isomorphic forms. In this section we will present the normalization algorithm and a proof that it always terminates.

### 5.2.1 Normalization Algorithm

The normalization algorithm is fairly straight forward application of the transformation rules in Section 3.5. We start by flattening the model down to its leaf level behaviors. Then, we apply channel resolution to replace channels with control and data dependencies. As a result, we have a simplified representation with only control and data dependencies. This representation is converted into a directed graph data structure like the graphical representation we have followed in visualizing MA expressions. This graph consists of 3 types of nodes, namely behavior, control and variable. There are two types of directed edges, namely control edges (between behaviors and control nodes) and data edges (between behaviors and variables or **from** variables to control nodes).

The next step is to clean up this graphical representation by removing any nodes that are unused in the model. For instance, if a behavior node is not reachable from the model's  $vsp$ , then the behavior will never be executed. This can be simply ensured by checking for any control relation that leads to the behavior in question. Similarly, if a variable does not have any readers or writers, it will not influence the execution results. Such a variable translates to a variable node without any incoming or outgoing edges and may be removed from our graph data structure. We now have a completely connected directed graph on which normalizing transformations may be applied.

---

**Algorithm 14** Normalize (MA\_Behavior  $M$ ) return MA\_Behavior  $M_n$

---

```

1:  $M' = \text{flatten}(M)$ 
2: for all  $c < a >: e_1 < out > \mapsto e_2 < in >$  do
3:    $M' = \text{ResolveLink}(M', a)$ 
4: end for
5: repeat
6:    $M_{old} = M'$ 
7:   for all  $e \in M'$  do
8:      $M' = \text{EliminateIdentity}(M', e)$ 
9:   end for
10:  for all  $q : \dots \& b_1 \& b_2 \& \dots \rightsquigarrow b \in M' \wedge (b_1 \triangleright b_2)$  do
11:     $M' = \text{RCDE}(M', b_1, b_2, b)$ 
12:  end for
13:  for all  $l : b_1 \rightsquigarrow e. l : b_2 \rightsquigarrow .q : \dots \& e \& \dots \rightsquigarrow b \in M' \wedge (vsp \triangleright b_1) \wedge (b \triangleright b_2) \wedge (b_2 \triangleright (b, b))$  do
14:     $M' = \text{Streamline}(M', b_1, b_2, e, b)$ 
15:  end for
16:  for all  $q : \dots \& b_1 \& \dots \rightsquigarrow b_2 \in M' \wedge (b_1 \triangleright b_2)$  do
17:     $M' = \text{RelaxControl}(M', q, b_1, b_2)$ 
18:  end for
19: until  $M_{old} == M'$ 
20:  $M_n = M'$ 

```

---

The pseudo code for the normalization process is given in Algorithm 14. We first apply the identity elimination rule on all possible identity behaviors that may be optimized away. Then we try to remove all extra control dependencies using the RCDE rule. This is followed by streamlining where ever applicable. Finally, we apply control relaxation rule on all applicable edges. The transformations are attempted all over again, in the same order, starting with identity elimination if there are any changes to the graph data structure during any of the transformations. The algorithm terminates if any round of transformations attempted modifies the graph.

### 5.3 Isomorphism Testing of Normalized Models

The normalized graphs of the original and refined model must be tested for isomorphism in order to establish their equivalence. During isomorphism checking, we consider key features of the nodes being compared. Specifically, two behavior nodes, each from different model, are considered a match if each has the same behavior type. We will assume that the name of the SpecC behavior type serves as the key. It can be seen that if two identical SpecC behaviors have different names, then their corresponding nodes will not match. However, we recall that during verifiable refinements, we do not touch the leaf level behaviors. Let us consider that we are testing for isomorphism of behaviors  $M$  and  $M'$ . Let  $b \in M$  and  $b' \in M'$  be matching behaviors. For simplicity, we will write this match as  $b \leftrightarrow b'$ . We will use the same convention for matching of variables and control conditions. It can be seen that if a model has more than one behavior instance of the same type, then a simple type comparison will not produce a one to one match. In such cases, we try to compare the neighboring control and variable nodes to establish one to one matching.

Matching of variables is ensured by checking for a match of their writer and reader behaviors. For instance, if variables  $v$  in  $M$  is connected to out-port  $p$  of  $b$ , and variable  $v'$  in  $M'$  is connected to out-port  $p$  of  $b'$  and we have  $b \leftrightarrow b'$ , then we can establish that  $v \leftrightarrow v'$ . Also, the variables should have matching reader behaviors. Specifically, if  $v$  in  $M$  is connected to in-port  $p_r$  of  $b_r$  and  $v'$  in  $M'$  is connected to in-port  $p_r$  of  $b'_r$ , where  $b_r \leftrightarrow b'_r$ , then  $v \leftrightarrow v'$ .

The matching of control conditions is slightly more complicated. In order for control condition nodes to match, they must have matching predecessors and successors and they must

have a matching boolean function. This means, that for a given condition  $q$  in model  $M$  and condition  $q'$  in model  $M'$ , such that

$$q : b_1 \& \dots \& b_n \rightsquigarrow b \in M, q = f(v_1, \dots, v_m) \text{ and}$$

$$q' : b'_1 \& \dots \& b'_n \rightsquigarrow b' \in M', q' = f'(v'_1, \dots, v'_m)$$

We claim that  $q \leftrightarrow q'$  iff

1.  $b \leftrightarrow b'$
2.  $\exists$  bijective mapping  $Pmatch : \{b_1, \dots, b_n\} \mapsto \{b'_1, \dots, b'_n\}$ , s.t.  $b_i Pmatch b'_j$ , iff  $b_i \leftrightarrow b'_j$
3.  $\forall i, 1 \leq i \leq m, v_i \leftrightarrow v'_i$
4.  $f = f' |_{\forall i, 1 \leq i \leq m, v'_i = v_i}$

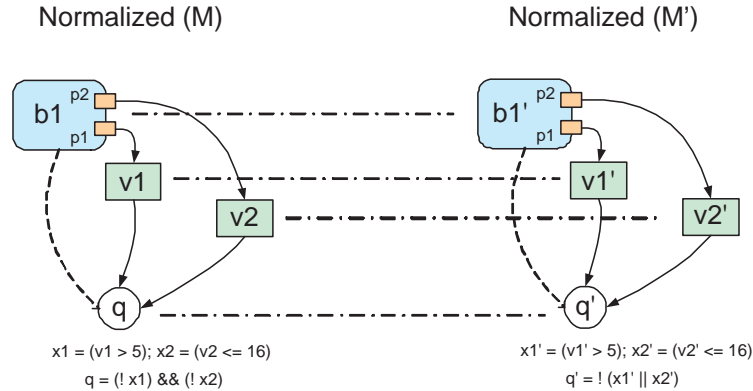


Figure 5.6: Matching graph nodes during isomorphism testing

An example of isomorphism matching for two normalized models is shown in Figure 5.6. The dashed lines across the figure show the matching of the individual nodes. Behaviors  $b$  in  $M$  and  $b'$  in  $M'$  are matched since they have the same type in SpecC. Variables  $v_1$  and  $v'_1$  are matching because they are connected to the same port of matching behaviors. Consider the two control conditions  $q$  and  $q'$  and the pairs of word-level predicates  $(x_1, x_2)$  and  $(x'_1, x'_2)$ . It can be seen that if we syntactically replace the variables in  $x'_1$  and  $x'_2$  by the corresponding matching variables in  $M$ , then the predicates become syntactically identical to  $x_1$  and  $x_2$ , respectively. After

this replacement, it is easy to see that the boolean functions for  $q$  and  $q'|_{x'_1=x_1, x'_2=x_2}$  are equal. Hence,  $q$  and  $q'$  can be matched.

In general, it is possible that we cannot get a unique matching for all nodes using the above method. If there are multiple matches for a given node, then the false matches can be iteratively removed by examining the neighbors in the graph. For example, if behavior  $b$  in  $M$  matches  $b'$  and  $b''$  in  $M'$ , then we examine the set of all the variables read by  $b$  in  $M$  and see if there exist matching variables read using the same ports by  $b'$  and  $b''$  in  $M'$ . Similar approach may be used for variables written by  $b$  and so on. The elimination process is repeated for variable nodes and control condition nodes until no more eliminations can be performed. At the end, if there exist node(s) in  $M$  that do not have any match(es) in  $M'$ , then we can conclude that  $M$  and  $M'$  could not be proven equivalent. If matches for all nodes are found then we conclude that  $M$  and  $M'$  are equivalent. Note again that the negative result may be false, if either the transformation rules are incomplete or the normalization process does not generate a proof.

It can be seen that for certain pathological cases, the iterative algorithm to find a match is inefficient. The problem we are trying to solve is in general a graph isomorphism problem, whose complexity has not yet been established to be either in P or NP. In fact, this special problem has its own complexity class [Tor00]. There exists several algorithms for matching two general graphs [CG70] that are more efficient than our brute force iterative method. However, heuristics like the type restrictions on the behaviors and the boolean functions for conditions significantly reduce the possible permutations. In all the models encountered in our experimental results, the matching process took only a few iterations to converge and give a result.

## 5.4 Experimental Results

A verification tool based on the technique in Section 5.2, was written in C++ for checking refinement of SpecC models. The functional abstraction module was used to derive the Model Algebraic representation from a SpecC Model. The normalization algorithm in Section 5.2.1 were then used to normalize this the algebraic representation. The normalized graphs for the original and refined model were compared using a simple graph isomorphism checker implemented within the verification tool. A BDD manipulation package CUDD [CUD] was used to maintain

Application	Refinement Type	Model before normalization	Normal Model	Total Number of Transformations	Verification Time (sec)
Voice Codec	Behavior Mapping	B:148, D:127, Q:161; CD:332, DD:429	B:89, D:127, Q:115; CD:332, DD:445	6131	3.3
		B:188, D:138, Q:202; CD:439, DD:453			
	Comm. Scheduling	B:188, D:138, Q:202; CD:439, DD:453		7065	3.7
		B:188, D:138, Q:202; CD:428, DD:453		7229	3.7
	Transaction Routing	B:188, D:138, Q:202; CD:428, DD:453			
		B:204, D:144, Q:221; CD:463, DD:470			
JPEG Encoder	Behavior Mapping	B:45, D:47, Q:56; CD:115, DD:104	B:27, D:47, Q:38; CD:109, DD:104	1602	1.6
		B:59, D:53, Q:68; CD:144, DD:113			
	Comm. Scheduling	B:59, D:53, Q:68; CD:144, DD:113		2740	2.1
		B:59, D:53, Q:68; CD:114, DD:113		2852	2.1
	Transaction Routing	B:59, D:53, Q:68; CD:114, DD:113			
		B:67, D:56, Q:78; CD:134, DD:119			

Figure 5.7: Performance of verification tool for various refinements

the boolean functions for the condition nodes.

The verification tool has been tested on a wide variety of applications and refinements. We present here, results from two applications from the multimedia domain. The first application is a GSM voice codec application [GZG99] ( 12K lines of SpecC code) for cellular phones. The second application is a JPEG encoder ( 3K lines of SpecC code). Three types of rearrangement refinements were tried for each application. The first refinement, *behavior mapping*, was used to move some behavior from one component to another. This refinement resulted in a different behavioral hierarchy and extra behaviors and synchronization to preserve the functionality of the design. The second refinement, *communication scheduling*, was used to reorder behavior execution inside components so as to change the bus traffic pattern. The final refinement, *transaction routing*, was used to change the bus architecture and route transactions via a transducer.

The table in Figure 5.7 shows results for the verification of aforementioned refinements on the two applications. The column *Model before normalization* gives statistics for the abstract functional model in terms of number of behavior nodes (B), variable nodes (D), condition nodes (Q), control dependency edges (CD) and data dependency edges (DD). The column *Normal Model* gives the statistics of the normalized graph. Note that normal form is unique for all models involved in the refinements. *Number of transformations* include transformations resulting from flattening, link resolution, identity elimination, control relaxation, redundant control dependency elimination (RCDE) and streamlining. We have also included miscellaneous transformations like

removal of unreachable nodes, variables without readers or writers etc. The number of transformations includes transformations performed for normalizing **both** the original model **and** the refined model. As we can see, the total verification time is in the order of a few seconds, which makes this technique very practical.

## 5.5 Chapter Summary

In the previous chapters we introduced Model Algebra as a system level modeling formalism and presented useful refinement algorithms that were proven to be functionality preserving. In this chapter, we used the theory of Model Algebra and the proof strategies for refinements to develop automatic methods for proving correctness of refinements. This technique is particularly useful since we can automatically verify if a certain refinement algorithm has been correctly implemented.

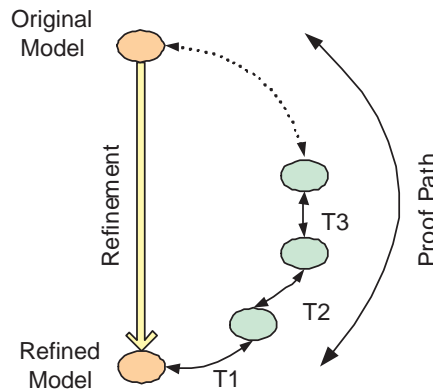


Figure 5.8: Refinement path vs. proof path

Figure 5.8 shows the automatic refinement strategy. The original model is refined and a new model is generated using a particular refinement algorithm. An alternate path from the original model to the refined model consists of a sequence of transformations, each being an instance of the transformation rules defined in MA. If the refinement and the transformations result in an identical model, then we have verified that the refinement is functionality preserving.

The implementation of the verification tool is targeted for system level models written in SpecC. We showed how the model algebraic expression for the original and refined SpecC models were derived using the functional abstraction module. Then the derived expressions were reduced

to a normal form by successively applying the transformation rules of MA. Finally, the normal expressions were compared using the isomorphism testing module of the tool.

## Chapter 6

# Conclusion

In this dissertation, we presented our work on the verification of some useful system level refinements. We introduced Model Algebra, a formalism that was used as a basis to reason about the construction, transformation and verification of system level models. It provided a theoretical foundation for representing and proving the correctness of system level refinements. We identified the commonly used refinements, namely behavior partitioning, scheduling and transaction routing and presented the syntactic modifications that result from such refinements. The refinements were then proven correct using sound transformation rules of Model Algebra. The refinement algorithms and their correctness proofs allow reliable implementations of system level refinements. Besides the theoretical results, we also presented the implementation of an automatic refinement verification tool that verifies if a refined model is functionally equivalent to the original model.

The complexity of modern embedded system designs forces designers to develop abstract performance models that simulate fast and give reasonably accurate metrics. Using these abstract models, the designer may evaluate various implementations and choose the most appropriate one. However, as we add details to the abstract model and gradually refine them, we must guarantee that the refined model is functionally equivalent to the original model. The problem of verifying equivalence of independently developed system level models is intractable at best. Therefore, for efficient verification, we proposed the verification of well defined refinements instead of arbitrary models themselves. We now look at the specific benefits of our work, the

contributions of this thesis and some ideas for future work.

## 6.1 Benefits

The key advantages of using Model Algebra and refinement verification in system-level design are enumerated as follows:

1. **No need to verify refined models:** The designer needs to perform property verification only on the first (most abstract) model. All models derived from this abstract model can be automatically verified using our refinement verification tool. Since property verification can be very time-consuming, we save significant amount of time by eliminating the need for re-verifying generated models.
2. **Correct automatic refinements:** A model refinement can be expressed as a sequence of syntactic transformations in Model Algebra. Since, each transformation is well defined and is proven to be sound, the refinement can be performed automatically and reliably.
3. **Automatic checking of model refinement:** Actual implementation of refinement tools may contain bugs that result in generated models that are not equivalent to the original model. The additional process of running the refinement verification tool gives the designer confidence that the refinement tool works correctly. If a mismatch is found, it can be used to debug the refinement tool.
4. **Disciplined system design:** Refinement verification works in a disciplined system design methodology, where detailed models are derived from abstract models, rather than being re-written. To get the advantage of our verification method, the system designers define clear semantics of models at each level of abstraction. Therefore, the design methodology enforces models that are not just simulatable, but also synthesizable and verifiable.

## 6.2 Contribution

The significance of our work is that we proposed, defined and proved through implementation our new concept of refinement verification in system-level design. The contribution of

our work can be further enumerated as follows.

1. We defined Model Algebra as a set of objects and composition rules. Expressions in Model Algebra can be used to represent system level models as a set of communicating hierarchical behaviors..
2. We defined formal execution semantics of models written in Model Algebra that allow us to reason about the functional equivalence of models.
3. We defined transformation rules for models written in Model Algebra and proved their soundness. This allowed us to syntactically transform one model in Model Algebra into a functionally equivalent model.
4. We identified useful system level refinements that may be used to evaluate different system implementations. These refinements were proved for correctness using above transformation rules. This allows development of reliable tools for automatic refinement of system level models.
5. Using the transformation rules and the proofs for refinement algorithms, we developed a tool for automatically checking if a certain refinement is implemented correctly. This allows us to have greater confidence in the model refinement results and the ability to debug the refinement tools.

### **6.3 Future Directions**

In this dissertation, we selected refinements namely behavior partitioning, scheduling and transaction routing. We used Model Algebra and its rules to verify algorithms for these refinements. In the future, we may identify more refinements that may be verified. There are several topics to be explored in this regard, some of which are enumerated below:

1. Verification of refinements such as slicing and packetizing of communicated data will require the extension of Model Algebra to include variable types and their compositions.
2. New transformation rules required for refinement to cycle accurate models.

3. Study of completeness (necessity and sufficiency) of the transformation rules.

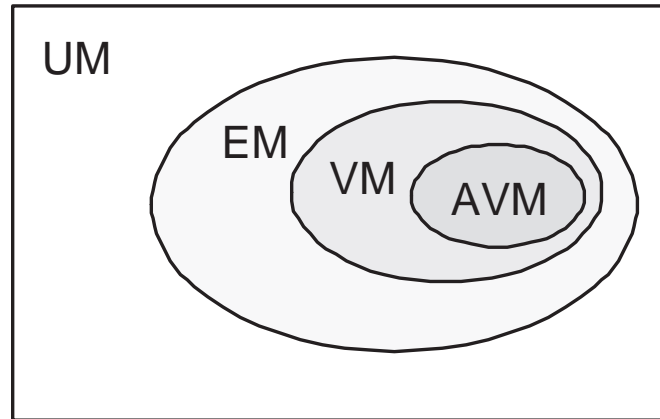


Figure 6.1: Classes of models in Model Algebra

The completeness aspect may be better understood from the illustration in Figure 6.1. The outermost rectangular box labeled  $UM$  is the set of all possible models that can be expressed in Model Algebra. It can be easily seen that this set is infinite. The universal set is divided into equivalence classes, one of which is labeled  $EM$ . All models in  $EM$  are equivalent with respect to our notion of trace equivalence discussed in Section 3.4. The set  $EM$  contains the set of verifiable models  $VM$ . These are equivalent models whose equivalence can be proved using the transformation rules of Model Algebra. Finally, the set  $VM$  contains the set of automatically verifiable models  $AVM$ . Models in this set are such that their equivalence can be proved using the transformation rules and there exists a tractable algorithm that can generate this proof. If  $EM = VM$ , then we say that the transformation rules are complete. This is a powerful result since equivalence, or the lack thereof, of any two models can be proved. If  $AVM = VM$ , then any proof of equivalence can be generated automatically in finite time.

# Bibliography

- [ABH<sup>+</sup>97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.
- [Acc01] Accellera C/C++ Working Group of the Architectural Language Committee. RTL Semantics, Draft Specification. February 2001.
- [AG03] Samar Abdi and Daniel Gajski. Formal Verification of Specification Partitioning. Technical Report ICS-TR-03-06, University of California, Irvine, March 2003.
- [AG04a] Samar Abdi and Daniel Gajski. Automatic generation of equivalent architecture model from functional specification. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 608–613, New York, NY, USA, 2004. ACM Press.
- [AG04b] Samar Abdi and Daniel Gajski. On deriving equivalent architecture model from system specification. In *ASP-DAC*, pages 322–327, 2004.
- [AG04c] Samar Abdi and Daniel Gajski. System Level Verification with Model Algebra. Technical Report ICS-TR-04-29, University of California, Irvine, October 2004.
- [AG05] Samar Abdi and Daniel D. Gajski. Functional validation of system level static scheduling. In *DATE*, pages 542–547, 2005.
- [BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. In *Information and Control*, pages 109–137, 1984.
- [BK87] Gastano Borriello and Randy Katz. Synthesis and optimization of interface transducer logic. In *Proceedings of the International Conference on Computer-Aided Design*, pages 274–277, November 1987.
- [BNP02] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computer*, pages 677–691, August 1986.

- [Bry90] Randal E. Bryant. Symbolic simulation - techniques and applications. In *Design Automation Conference*, pages 517–521, 1990.
- [Cam90] R. Camposano. Behavior-preserving transformations for high-level synthesis. In *Proceedings of the Mathematical Sciences Institute workshop on Hardware specification, verification and synthesis: mathematical aspects*, pages 106–128. Springer-Verlag New York, Inc., 1990.
- [CG70] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.
- [CG03] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press.
- [CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [CGP00] E.M. Clarke, O. Grumberg, and DA Peled. *Model Checking*. MIT Press, 2000.
- [CHBW03] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Case studies of model checking for embedded system designs. In *Third International Conference on Application of Concurrency to System Design*, pages 20–28, June 2003.
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
- [CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 368–371, New York, NY, USA, 2003. ACM Press.
- [CoW] CoWare N2C. Available: <http://www.coware.com/cowareN2C.html>.
- [CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M.K. Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222, September 1994.
- [CUD] CU Decision Diagram Package. Available: <http://vlsi.colorado.edu/fabio/CUD-D/cuddIntro.html>.
- [Cyn] Cynlib[online]. Available: <http://www.cynapps.com/>.
- [Don04] Adam Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS*, pages 75–80, 2004.

- [Dre04] Rolf Drechsler. Towards formal verification on the system level. In *International Workshop on Rapid System Prototyping*, pages 2–5, June 2004.
- [eve] e Verification language[online]. Available: <http://www.cadence.com/>.
- [FH05] Xiushan Feng and Alan J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 307–316, New York, NY, USA, 2005. ACM Press.
- [Fok00] Wan Fokkink. *Introduction to Process Algebra*. Springer, January 2000.
- [For] Cynthesizer[online]. Available: <http://www.forte.com/>.
- [GABP98] Guy Gogniat, Michel Auguin, Luc Bianco, and Alain Pegatoquet. Communication synthesis and hw/sw integration for embedded system design. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 35–98, March 1998.
- [Gaj88] Daniel Gajski. *Silicon Compilation*. Addison Wesley, 1988.
- [Gaj97] Daniel Gajski. *Principles of Digital Design*. Prentice Hall, 1997.
- [Gan01] Jack Ganssle. Ice technology unplugged. *Embedded Systems Programming*, page 103, October 2001.
- [GDGP02] Daniel Gajski, Rainer Domer, Andreas Gerstlauer, and Junyu Peng. *System Design with SpecC*. Kluwer Academic Publishers, January 2002.
- [GDLW92] Daniel Gajski, Nikil Dutt, S. Lin, and Allen Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
- [GG02] Andreas Gerstlauer and Daniel D. Gajski. System-level abstraction semantics. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 231–236, New York, NY, USA, 2002. ACM Press.
- [GL97] Rajesh K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, April 1997.
- [GLM<sup>+</sup>02] Ferid Gharsalli, D. Lyonnard, Samy Meftali, F. Rousseau, and Ahmed A. Jerraya. Unifying memory and processor wrapper architecture in multiprocessor soc design. In *Proceedings of the International Symposium on System Synthesis*, Oct 2002.
- [Gor88] Mike Gordon. *Specification and verification*, 1988.
- [GPB01] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

- [GVNG94] Daniel Gajski, Frank Vahid, Sajiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.
- [GYG03] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. Rtos modeling for system level design. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 10130–10135, Washington, DC, USA, 2003. IEEE Computer Society.
- [GYJ01a] Lovic Gauthier, Sunjoo Yoo, and Ahmed Jerraya. Automatic generation and targeting of application specific operating systems and embedded system software. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 679–685, March 2001.
- [GYJ01b] Lovic Gauthier, Sunjoo Yoo, and Ahmed Jerraya. Automatic generation of application specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the Design Automation Conference*, pages 518–523, June 2001.
- [GZD<sup>+</sup>00] Daniel Gajski, Jiwen Zhu, Rainer Domer, Andreas Gerstlauer, and Suqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.
- [GZG99] Andreas Gerstlauer, Shuqing Zhao, and Daniel Gajski. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, February 1999.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [Hol97] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), June 1997.
- [Inc98] Texas Instruments Inc. TMS32C5x User’s Guide, June 1998.
- [Inc00] ARM Inc. ARM9TDMI Technical Reference Manual, March 2000.
- [Inc01] Motorola Inc. ColdFire CF4e Core User’s Manual, June 2001.
- [Inv] Inventra. Inventra Core library.
- [JEK<sup>+</sup>90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking:  $10^{20}$  States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
- [JK97] Jens Jorgensen and Lars Kristensen. Verification of colored petri nets using state spaces with equivalence classes. In *Proceedings of the Workshop on Petri Nets in System Engineering*, pages 20–31, September 1997.

- [Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Info. Proc.*, pages 471–475, August 1974.
- [KCY03] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
- [LDR01] Kanishka Lahiri, Sujit Dey, and Anand Raghunathan. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In *VL-SID '01: Proceedings of the The 14th International Conference on VLSI Design (VL-SID '01)*, page 29, Washington, DC, USA, 2001. IEEE Computer Society.
- [LJ01] Jiong Luo and Niraj K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Design Automation Conference*, pages 444–449, 2001.
- [LRL01] Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. LOTTERY-BUS: A new high-performance communication architecture for system-on-chip designs. In *Design Automation Conference*, pages 15–20, 2001.
- [LV94] Bill Lin and Steven Vercauteren. Synthesis of concurrent system interface modules with automatic protocol conversion generation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 101–108, November 1994.
- [Mat] Matlab. Available: <http://www.mathworks.com/>.
- [MDR<sup>+</sup>00] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication scheduling. *SIGPLAN Not.*, 35(11):82–92, 2000.
- [Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [Mid96] Middlehoek. A methodology for the design of guaranteed correct and efficient digital systems. In *IEEE International High Level Design Validation and Test Workshop*, November 1996.
- [Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.
- [Mil99] R. Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, 1999.
- [Mis01] International Technology Roadmap for Semiconductors, <http://public.itrs.net/>, 2001.
- [M.J88] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

- [MMZ<sup>+</sup>01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
- [NG95] Sanjiv Narayan and Daniel Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the Design Automation Conference*, pages 468–473, June 1995.
- [PAG02] Junyu Peng, Samar Abdi, and Daniel Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 332–337, January 2002.
- [PCPK00] Bong-Il Park, Hoon Choi, In-Cheol Park, and Chong-Min Kyung. Synthesis and optimization of interface hardware between ip's operating at different clock frequencies. In *Proceedings of the International Conference on Computer Design*, pages 519–524, June 2000.
- [Pet81] J.L. Peterson. *Petri Net Theory and Modeling of Systems*. Prentice Hall, 1981.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.
- [PRSV98a] Passerone, James A. Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the International Conference on Computer-Aided Design*, pages 437–444, November 1998.
- [PRSV98b] Roberto Passerone, James A. Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the Design Automation Conference*, pages 8–13, June 1998.
- [PVS] PVS[online]. Available: <http://pvs.csl.sri.com/>.
- [Rab96] Jan M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.
- [Raj95] Sreeranga Rajan. Correctness of transformations in high level synthesis. In *International Conference on Computer Hardware Description Languages and their Applications*, pages 597–603, June 1995.
- [RSV97] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface based design. In *Proceedings of the Design Automation Conference*, pages 178–183, June 1997.
- [SB92] J. Sun and R. Brodersen. Design of system interface modules. In *Proceedings of the International Conference on Computer-Aided Design*, pages 478–481, November 1992.

- [SBR05] Jürgen Schnerr, Oliver Bringmann, and Wolfgang Rosenstiel. Cycle accurate binary translation for simulation acceleration in rapid prototyping of socs. In *DATE*, pages 792–797, 2005.
- [SDN87] T. Ma S. Devadas and R. Newton. On the verification of sequential machines at different levels of abstraction. In *Proceedings of the Design Automation Conference*, pages 271–276, June 1987.
- [SG01] Dongwan Shin and Daniel Gajski. Internal Representation for RTL Design Methodology. Technical Report ICS-TR-01-50, University of California, Irvine, June 2001.
- [SG02] Dongwan Shin and Daniel Gajski. Queue Generation Algorithm for Interface Synthesis. Technical Report ICS-TR-02-03, University of California, Irvine, February 2002.
- [SJ99] I. Sander and A. Jantsch. Formal system design based on the synchrony hypothesis, functional models and skeletons. In *VLSID '99: Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, page 318, Washington, DC, USA, 1999. IEEE Computer Society.
- [SJ02] Ingo Sander and Axel Jantsch. Transformation based communication and clock domain refinement for system design. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 281–286, New York, NY, USA, 2002. ACM Press.
- [SJL03] Ingo Sander, Axel Jantsch, and Zhonghai Lu. Development and application of design transformations in forsyde. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10364, Washington, DC, USA, 2003. IEEE Computer Society.
- [sle] SLEC[online]. Available: <http://www.calypto.com/>.
- [SM98] James Smith and Giovanni De Micheli. Automated composition of hardware components. In *Proceedings of the Design Automation Conference*, pages 14–19, June 1998.
- [SM02] Robert Siegmund and Diermar Muller. A novel synthesis technique for communication controller hardware from declarative data communication protocol specifications. In *Proceedings of the Design Automation Conference*, pages 602–607, June 2002.
- [SOS<sup>+</sup>02] Hiroshi Saito, Takaya Ogawa, Thanyapat Sakunkonchak, Masahiro Fujita, and Takashi Nanya. An equivalence checking methodology for hardware oriented c-based specifications. In *IEEE International High Level Design Validation and Test Workshop*, pages 274–277, October 2002.
- [SS98] D.R. Surma and Edwin Sha. Collision graph based communication scheduling for parallel systems. *International Journal of Computers and Their Applications.*, 5(1), March 1998.

- [Sysa] SystemC, OSCI[online]. Available: <http://www.systemc.org/>.
- [Sysb] SystemVerilog[online]. Available: <http://www.systemverilog.org/>.
- [Tor00] J. Toran. On the hardness of graph isomorphism. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 180–188, Washington, DC, USA, 2000. IEEE Computer Society.
- [vdB94] M. von der Beeck. A comparison of statechart variants. In *Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863*, pages 128–148, 1994.
- [vGG89] Rob van Glabbeek and Ursula Goltz. Equivalence notions for concurrent systems and refinement of actions. *Lecture Notes in Computer Science; Mathematical Foundations of Computer Science 1989*, 379:237–248, 1989.
- [VT97] Frank Vahid and L. Tauro. An object-oriented communication library for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 81–86, March 1997.
- [WM03] Shaojie Wang and Sharad Malik. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of the International Symposium on System Synthesis*, pages 37–44, September 2003.
- [WTCV00] Chun Wong, Filip Thoen, Francky Catthoor, and Diederik Verkest. Static task scheduling of embedded systems. In *3rd Workshop on System Design Automation - SDA 2000 Rathen, Germany*, pages 23–30, March 2000.
- [YGG03] Haobo Yu, Andreas Gerstlauer, and Daniel Gajski. Rtos scheduling in transaction level models. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 31–36, New York, NY, USA, 2003. ACM Press.
- [ZSY<sup>+</sup>00] Pei Zhang, Dongwan Shin, Haobo Yu, Qiang Xie, and Daniel Gajski. SpecC RTL Design Methodology. Technical Report ICS-TR-00-44, University of California, Irvine, December 2000.