UNIVERSITY OF CALIFORNIA, SAN DIEGO

High-Level Verification of System Designs

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

in

Computer Science and Engineering

by

Sudipta Kundu

2009

DEDICATION

To the two ladies of my life – my mother and my wife.

EPIGRAPH

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

—Isaac Asimov

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Acknowledgement is not a mere formality but a genuine opportunity to express the sincere thanks to all those; without whose active support and encouragement this thesis would not have been successful.

I express my deep sense of regards and indebtedness to my advisers Prof. Rajesh Gupta and Prof. Sorin Lerner for their valuable guidance, continuous encouragement and wholehearted support, which are of immense help to me in completing this thesis. In spite of their hectic schedule they have always extended their help and invaluable suggestions.

I express my thanks to my committee members Prof. Ranjit Jhala, Prof. Ingolf Krueger, and Prof. Bill Lin and to all the faculty members and the staff of the Department of Computer Science and Engineering for their continuous help and support.

My sincere thanks to Dr. Malay Ganai, NEC Laboratories America, Princeton, NJ for his continuous encouragement and extremely valuable insights for research. I also thank Chao Wang, Nishant Sinha, Aarti Gupta, and all other members of the System LSI and Software Verification group at NEC Laboratories America for uncountably many interesting discussion and explanation of key concepts in Verification.

I thank Prof. Alan J. Hu for his valuable insights and comments on the translation validation part of this thesis.

My thanks to my friends Federic Doucet, Zachary Tatlock and other members of the MESL and Progsys group for their ceaseless effort, constant encouragement, and also for the endless good times during the making of this dissertation.

Words are not enough to express my indebtedness and gratitude toward my father Prof. S. C. Kundu and my wife Shreya, to whom I owe every success and achievements of my life. Their constant support and encouragement under all odds has brought me where I stand today. Also, my love and respect toward my brother Sumanta and sister-in-law Sharon for always believing in me.

Finally, I thank all my friends for their love, cheerful encouragements and valuable criticism.

Papers included in this dissertation

Chapter 2, in part, have been accepted for publication as “High-Level Verification” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *IPSJ Transactions on*

System LSI Design Methodology [KLG09]. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been published as “Partial Order Reduction for Scalable Testing of SystemC TLM Designs” by Sudipta Kundu, Malay Ganai and Rajesh Gupta in *DAC 08: Proceedings of the 45th annual conference on Design Automation* [KGG08]. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, have been accepted for publication as “Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions” by Sudipta Kundu and Malay Ganai in *SPIN 09: Proceedings of the 16th International SPIN Workshop on Model Checking of Software* [GK09]. The dissertation author was the primary investigator and author of the parts of this paper that is included in this dissertation.

Chapter 5, in part, is in preparation for publication as “Efficient Symbolic Analysis for Concurrent Programs” by Sudipta Kundu and Malay Ganai. The dissertation author is the primary investigator and author of this paper.

Chapter 6, in part, has been published as “Automated Refinement Checking of Concurrent Systems” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *ICCAD 07: Proceedings of the 2007 IEEE/ACM International Conference on Computer Aided Design* [KLG07]. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, has been published as “Validating High-Level Synthesis” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *CAV 08: Proceedings of the 20th international conference on Computer Aided Verification* [KLG08]. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, is in preparation for publication as “Translation Validation of High-Level Synthesis” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta. The dissertation author is the primary investigator and author of this paper.

Chapter 7, in part, have been accepted for publication as “Proving Optimizations Correct using Parameterized Program Equivalence” by Sudipta Kundu, Zachary Tatlock, and Sorin Lerner in *PLDI 09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation* [KTL09]. The dissertation author was the primary investigator and author of the parts of this paper that is included in this dissertation.

Curriculum Vitae

Education

Ph.D., Computer Science and Engineering, Sept 2009
University of California at San Diego (UCSD), La Jolla, CA
Thesis: *High-Level Verification of System-Level Designs*
Advisers: Prof. Rajesh Gupta and Prof. Sorin Lerner
GPA: 3.88/4.0

5 Year Integrated Master of Science, Mathematics and Computing, May 2004
Indian Institute of Technology (IIT-KGP) at Kharagpur, India
M.Sc. Thesis: *Design and Optimization of an Application Specific Operating System*
Advisers: Prof. Anupam Basu and Prof. U. C. Gupta
B.Sc. Thesis: *Assembler and Simulator for a subset of 8086 Assembly Language*
Adviser: Dr. Pawan Kumar
GPA: 9.12/10.0

Honors

CAL-IT(2) Fellowship award, 2004-05 from CSE Department, UCSD.
Silver Medal, 2004 from IIT, Kharagpur for being ranked first in the department.
Student Fellowship, 17th International Conference on VLSI Design, Jan 2004, India.
J. C. Ghosh memorial prize, 2003 from IIT Kharagpur for academic excellence.

Research Experience

Summer Internships

2008: *NEC Labs America, Princeton, NJ*; "Verification of Concurrent Programs using a Bounded Model Checking (BMC) Framework" under the supervision of *Dr. Malay Ganai.*

2007: *NEC Labs America, Princeton, NJ*; "Verification of SystemC Transaction Level Model (TLM)" under the supervision of *Dr. Malay Ganai.*

2006: *INRIA, Rennes, France*; "Compositional Modeling and Synthesis of Systems on a Chip" under the supervision of *Prof. Jean-Pierre Talpin.*

2005: *INTEL Corporation, Portland, OR*; "Dynamic Measurement of Available Bandwidth in a Heterogeneous Home Network", under the supervision of *Narasimham Gadiraju.*

2004: *Communication Empowerment Laboratory, IIT Kharagpur*; "Porting of Embedded Linux and Applications to TI OMAP Platform", under the guidance of *Prof. Anupam Basu.*

2002: *Centre for Internet Research, National University of Singapore* (NUS); "Process Management of Embedded Linux" under the guidance of *Dr. A. L. Ananda.*

External Reviewer

IEEE Transactions on Computer Aided Design.
IEEE Transactions on Design Automation of Electronic Systems.
IEEE Embedded Systems Letters.

Publications

Journal Papers

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Translation Validation of High-Level Synthesis. *In Preparation*, 2009.

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. High-Level Verification. ***IPSJ Transactions*** *on System LSI Design Methodology*, 2009. (Invited Paper).

Conference Papers

Chao Wang, **Sudipta Kundu**, Malay Ganai, and Aarti Gupta. Symbolic Predictive Analysis for Concurrent Programs. In ***FM '09****: Proceedings of the 16th International Symposium on Formal Methods*, 2009.

Malay Ganai and **Sudipta Kundu**. Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In ***SPIN '09****: Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, 2009.

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct using Parameterized Program Equivalence. In ***PLDI '09****: Proceedings* *of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating High-Level Synthesis. In ***CAV '08****: Proceedings of the 20th international conference on Computer Aided Verification*, pages 459–472, Princeton, NJ, USA, 2008. Springer.

Sudipta Kundu, Malay Ganai, and Rajesh Gupta. Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In ***DAC '08****: Proceedings of the 45th annual conference on Design Automation*, pages 936–941, New York, NY, USA, 2008. ACM.

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Automated Refinement Checking of Concurrent Systems. In ***ICCAD '07****: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 318–325, NJ, USA, 2007. IEEE Press.

Gurashis Singh Brar, **Sudipta Kundu**, Pratik Worah, Susmit Biswas, Arijit Mukherjee, and Anupam Basu. OaSis: An Application Specific Operating System for an Embedded Environment. In ***VLSI Design '04****: 17th International Conference on VLSI Design*, pages 776–779, Mumbai, India, 2004. IEEE Press.

Patents

Malay Ganai and **Sudipta Kundu**. Partial Order Reduction for Scalable Testing in System Level Design. *US Patent Pending*, 2008.

ABSTRACT OF THE DISSERTATION

High-Level Verification of System Designs

by

Sudipta Kundu

Doctor of Philosophy in Computer Science and Engineering

University of California, San Diego, 2009

Professor Rajesh K. Gupta, Chair
Professor Sorin Lerner, Co-Chair

Given the growing size and heterogeneity of Systems on Chip (SOC), the design process from initial specification to chip fabrication has become increasingly complex. The growing complexity provides incentive for designers to use high-level languages such as C, SystemC, and SystemVerilog for system-level design. While a major goal of these high-level languages is to enable verification at a higher level of abstraction, allowing early exploration of system-level designs, the focus so far has been on traditional testing techniques such as random testing and scenario-based testing.

This dissertation focuses on *high-level verification of system designs*. We envision a design methodology that relies upon advances in synthesis techniques as well as on incremental *refinement* of the design process. These refinements can be done manually or through elaboration tools. Our work addresses verification of specific properties in high-level languages as well as checking that the refined implementations are equivalent to their high-level specifications. The novelty of each of these techniques is that they use a combination of formal techniques to do *scalable* verification of system designs completely *automatically*.

Our work falls into two categories: (a) methods for verifying properties of high-level designs and (b) methods for verifying that the translation from high-level design to a low-level Register Transfer Language (RTL) design preserves semantics. Taken together, these two parts guarantee that properties verified in the high-level design are preserved through the translation to low-level RTL. By performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, we expand hardware development methodology to provide strong and expressive guarantees that are difficult to achieve by directly analyzing the low-level RTL code. Our techniques for high-level verification have been implemented in a framework, which consists of four tools, namely Satya, Candor, Surya, and PEC. We demonstrate the value of our techniques by verifying various industrial strength designs and a complex CAD-tool package called Spark.

Chapter 1

Introduction

The quantitative changes brought about by Moore's law in design of integrated circuits (ICs) affect not only the scale of the designs, but also the scale of the process to design and validate such chips. While designer productivity has grown at an impressive rate over the past few decades, the rate of improvement has not kept pace with chip capacity growth leading to the well known design-productivity-gap [Kah01]. The problem of reducing the design-productivity-gap is crucial in not only handling the complexity of the design, but also combating the increased fragility of heterogeneous components that are composed in a design. Unlike software programs, integrated circuits are not repairable. The development costs are so high that multiple design spins are ruled out, a design must be correct in the one and often the only one design iteration to implementation.

High-Level Synthesis (HLS) [GDWL92, Mic94, Mic90, WC91, Lin97, GR94, GDGN03, GB08] is often seen as a solution to bridge the design-productivity-gap. HLS is the process of generating the Register Transfer Level (RTL) design consisting of a data path and a control unit from the behavioral description of a digital system, expressed in languages like C, C++ and Java. The synthesis process consists of several inter dependent sub-tasks such as: specification, compilation, scheduling, allocation, binding and control generation. HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [WC91, Lin97, GDGN03, GB08]. This shift in design paradigm enables designers to avoid many low-level design issues early in the design process. It also enables early design space exploration, and faster functional verification time. However, for verification of high-level designs, the focus so far has been on traditional testing techniques such as random testing and scenario-based testing. Over the last few decades we have seen many unfortunate examples of hardware bugs (like Pentium FDIV bug, Killer poke, and Cyrix coma bug) that have eluded testing techniques. Recently, many techniques inspired from *formal* methods have emerged as an alternative to ensure the correctness of these high-level designs, overcoming some of the limitations of traditional testing techniques.

The new techniques and methodology for verification and validation at higher level of abstraction are collectively called *high-level verification* techniques. The high-level verification problem can be further divided into two parts. The first part deals with verifying properties of high-level designs. The methods for verifying high-level designs allow designers to check for certain properties such as functional behavior, absence of deadlocks and assertion violations in their designs. Once the properties are checked, the designers refine their design to low-level RTL manually or using a HLS tool. HLS tools are large and complex software systems, and as such they are prone to logical and implementation errors. Errors in these tools may lead to the synthesis of RTL designs with bugs in them. As a result, the second part deals with verifying that the translation from high-level design to low-level RTL preserves semantics. Taken together, these two parts guarantee that properties satisfied by the high-level design are preserved through the translation to low-level RTL.

Unfortunately, despite significant amount of work in the area of formal verification we are far from being able to prove automatically that a given design always does the right thing, or a given synthesis tool always produces target programs that are semantically equivalent to their source versions. However, with recent advances in SAT solvers [MMZ$^+$01, GN07], automated theorem proving [ORS92, Gor88, Pau94], and model checking [BCM$^+$90, COYC03, TVGSV95] researchers are at least able to prove that the designs and tools satisfy important properties. Also, in many cases they are able to guarantee the functional equivalence between the initial behavioral description and the RTL output of the HLS process.

In this thesis, we envision a design methodology that relies upon advances in synthesis techniques as well as on incremental refinement of design process. These refinements can be done manually or through elaboration tools. Our work addresses verification of specific properties in high-level languages as well as checking that the refined implementations are equivalent to their high-level specifications. While experience shows

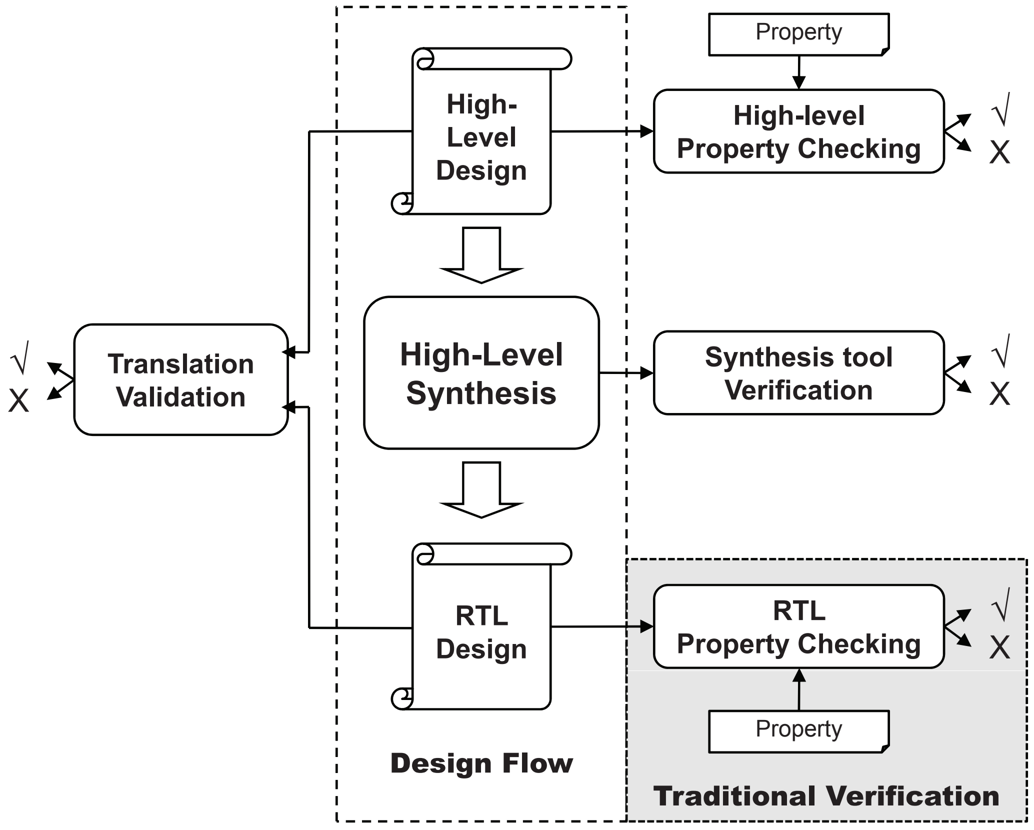


Figure 1.1: Overview of high-level verification

that no single technique (including the ones we developed specifically for high-level verification) can be universally useful, we have found that an intelligent combination of a number of these techniques driven by well considered heuristics is likely to prove parts of a design or tool correct, and also in many cases find bugs in them. The key contribution of this thesis is that it explores a combination of formal techniques to do *scalable* verification of system designs completely *automatically*.

1.1 Overview of High-Level Verification

The HLS process consists of performing stepwise transformations from a behavioral specification into a structural implementation (RTL). The main benefit of HLS is that it provides faster time to RTL and faster verification time. Figure 1.1 shows the various components involved in high-level verification and how they interact. The design flow from high-level specification to RTL is shown along with various verification tasks. These tasks can be broadly classified as follows:

1. High-level property checking
2. Translation validation
3. Synthesis tool verification
4. RTL property checking

Traditionally, designers start their verification efforts directly for RTL designs. However, with the popularity of HLS, these efforts are moving more toward their high-level counterparts. This is particularly interesting because it allows faster (sometimes by three orders of magnitude [Var07]) functional verification time, when compared to a more detailed low-level RTL implementation. Furthermore, it enables more elaborate design space exploration, which in turn leads to better quality of design. Since RTL property checking techniques have been widely explored in earlier works [Gup92, KG99, McF93a], here we focus only on the first three verification tasks.

The first category of methods, *high-level property checking*, allow various properties to be verified on the high-level designs. Once the important properties that the high-level components need to satisfy have been checked, various other techniques are used in order to prove that the translation from high-level design to low-level RTL is correct, thereby also guaranteeing that the important properties of the components are preserved.

The second category *translation validation* include techniques that try to show, for each translation that the HLS tool performs, that the output program produced by the tool has the same behavior as the original program. Although this approach does not guarantee that the HLS tool is bug free, it does guarantee that any errors in translation will be caught when the tool runs, preventing such errors from propagating any further in the hardware fabrication process.

The third category *synthesis tool verification* consists of techniques whose goal is to prove automatically that a given optimizing HLS tool itself is correct. Although, these techniques have same goal as translation validation i.e., to guarantee that a given HLS tool produces correct result, these techniques are different because they can prove the correctness of parts of the HLS tool *once and for all*, before they are ever run.

In this thesis, we have explored techniques for each one of the three areas outlined above, namely high-level property checking, translation validation, and synthesis tool verification. In the following section we briefly describe our techniques from each of these areas, outlining the connections and trade-offs between them.

1.2 Overview of Our Approach

As mentioned in the previous section our work falls into three categories. The key insight behind our approach is that by performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, we expand hardware development methodology to provide strong and expressive guarantees that are difficult to achieve by directly analyzing the low-level RTL code. In the following subsections we briefly discuss each of the techniques we developed specifically for high-level verification.

1.2.1 High-Level Property Checking

Starting with a high-level design, we use *model checking* techniques to verify that the design satisfies a given property such as the absence of deadlocks or assertion violations. Model checking in its pure form suffers from the well-known *state explosion* problem. To cope with this problem, some systems give up completeness of the search and focus on the bug finding capabilities of model checking. This line of thought lead to execution-based model checking approach, which for a given test input and depth, systematically explores all possible behaviors of the design (due to asynchronous concurrency). The most striking benefit of execution-based model checking approach is that it can analyze feature-rich programming languages like C++, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this approach is the idea of *stateless* search, meaning it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. To address this, one can use dynamic *partial-order-reduction* (POR) techniques to avoid generation of two paths that have the same effect on the design's behavior. Intuitively, POR techniques exploit the independence between parallel threads to search a reduced set of paths and still remain provably sufficient for detecting deadlocks and assertion violations.

We implemented Satya (Chapter 4), a novel *query-based* model checking tool that combines static and dynamic POR techniques along with high-level semantics of SystemC to intelligently explore all possible behaviors of a SystemC design. We reduce the runtime overhead by computing the dependency information statically and using it during runtime, without significant loss of precision. In our experiments Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), which may not have been found by simulation.

Another approach for model checking is to use symbolic algorithms that manipulate sets of states instead of individual states. These algorithms avoid ever building the complete state graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. *Bounded Model Checking* (BMC) is one such algorithm that unrolls the control flow graph (loop) for a fixed number of steps (say k) and checks whether a property violation can occur in k or fewer steps. This typically involves encoding the bounded model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT or SMT (Satisfiability Modulo Theory) solver. A key challenge for BMC is to generate efficient verification conditions that can be easily solved using the appropriate solver.

We developed a new symbolic method (Chapter 5), which combines POR with an asynchronous modeling approach that generate verification conditions directly without an explicit scheduler. We introduce the notion of *Mutually Atomic Transactions* (MAT): two transactions are mutually atomic when there exists exactly one conflicting shared-access pair between them and it appears at the end of the transactions. Previous approaches add interleaving constraints between all pairwise global accesses, thereby allowing redundant interleavings. We reduce the verification conditions by allowing pairwise interleaving constraints *only* between MATs. To evaluate our approach we implemented our algorithms in a tool called Candor. Our experimental results show that our approach improves the current state of the art both in performance and in size of the verification condition [GK09].

1.2.2 Translation Validation

Once the important properties of the high-level components have been verified, the translation from the high-level design to low-level RTL still needs to be proven correct, thereby guaranteeing that the important properties of the components are preserved. One approach to prove that the translation from high-level design to low-level RTL is correct is to show – for each translation that the HLS tool performs – the output program produced by the tool has the same behavior as the original program.

We developed a translation validation algorithm (Chapter 6) that uses a *bisimulation relation* approach to automatically prove the equivalence between two concurrent systems. We implemented our algorithm in a system called Surya and used it to validate the synthesis process of Spark [GDGN03], a parallelizing HLS framework. Surya validates all the code transformation phases (except for parsing, binding and code generation) of Spark against the initial behavioral description. Furthermore, our experiments showed that with only a fraction of the development cost of Spark, our algorithm can validate the translations performed by Spark, and it even uncovered two previously unknown bugs that eluded testing and long-term use.

1.2.3 Synthesis Tool Verification

Another approach to guarantee correctness of the translation from high-level design to low-level RTL, is by proving the HLS tool itself correct (Chapter 7). Unlike translation validation, this approach proves the correctness of an HLS tool *once and for all*, before it is ever run. Because some of the most error prone parts of an HLS tool are its optimizations, we developed a technique that proves the correctness of optimizations using *Parametrized Equivalence Checking* (PEC) [KTL09]. Furthermore, our approach is not limited to only HLS tools; it can be used for any domain that transforms an input program using semantics-preserving optimizations, such as optimizers, compilers, and assemblers.

The PEC technique is a generalization of translation validation that proves the equivalence of *parameterized programs*. A parameterized program is a partially specified program that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not modify certain variables. To highlight the power of PEC, we designed a language for implementing complex optimizations using many-to-many rewrite rules, and used this language to implement a variety of optimizations including software pipelining, loop unrolling, and loop unswitching. Using our PEC implementation, we were able to automatically verify that all the optimizations we implemented in our language preserve program behavior.

1.3 Contributions

The primary contribution of this thesis is to explore various formal techniques that can be used for high-level verification. We believe by performing verification on the high-level design, and then checking that all refinement steps are correct, the domain of high-level verification can provide strong and expressive guarantees that would have been difficult to achieve by directly analyzing the low-level RTL code. Our goal is to move the functional verification tasks earlier in the design phase, thereby allowing faster verification time and possibly quicker time to market.

To systematically explore the domain of high-level verification, we classified the various verification tasks into three main parts, namely high-level property checking, translation validation, and synthesis tool verification. We developed techniques for each of the above mentioned verification tasks. The novelty of our approaches is that it combines a number of formal techniques along with well considered heuristics to do *scalable* verification of high-level designs completely *automatically.*

To evaluate our approaches we developed a high-level verification framework that consists of four tools and complemented methodology to use these tools. The four tools are: Satya and Candor for high-level property checking, Surya for translation validation, and PEC for synthesis tool verification. These tools use state-of-the-art techniques from areas such as model checking, theorem proving, satisfiability modulo theories, static analysis and compiler correctness. Apart from these techniques our framework exploits structures specific to high-level designs, thereby in many cases simplifying our algorithms and improving their performance. For example, Satya exploits SystemC specific semantics to efficiently explore a reduced set of possible executions, and Surya relies on structure preserving transformations that are predominantly used in HLS.

Our framework enables experimentation with large "real-world" designs and tools. The key characteristics of our approach to high-level verification are as follows:

Scalable: Most of our analyses are modular as they work on one entity at a time. For example, Surya works on one procedure at a time, and PEC works on one transformation at a time. Furthermore, in the cases where we have to analyze the entire design together we apply various reduction techniques like partial-order reduction, context bounding, and program structure based reduction (e.g. lock-unlock, fork-join, wait-notify, etc.). While these software engineering decisions and reductions theoretically limits the scope of the verification tasks on a given design, it is rarely an issue in practice as it follows the designer's and tool developer's programming abstractions.

Practical: Our tools are practical enough to be applied to industrial strength designs and tools. In particular, we used Satya to check an industrial benchmark namely the TAC platform [Mic05], and used Surya to validate the synthesis process of Spark [GDGN03], a state-of-the-art academic HLS framework. We compared the results obtained using our tools with previous approaches. For example, in our experiments Candor outperforms previous approaches [GG08, WYKG08] in most cases, both in performance and size of the verification problem.

Useful: Our tools are able to automatically guarantee the correctness of various properties and transformations. Apart from correctness guarantee, these tools are also quite useful for finding bugs. For example, Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), and Surya was able to uncover two previously unknown bugs in the Spark HLS framework.

1.4 Thesis Organization

The organization of this thesis is shown in Figure 1.2. Chapter 2 presents a detailed discussion of the related works in the area of high-level verification. In particular, we divide the related works in to three main areas, namely high-level property checking, translation validation, and synthesis tool verification. We discuss the various tools and techniques explored in these areas. Apart from these we again discuss some more related works specific to the chapters in each of the following chapters, where we compare our approach with the existing approaches.

In Chapter 3, we present a brief overview of the three different parts of high-level verification on which our approaches are applied. More specifically, we present a brief introduction of high-level designs, RTL designs, and high-level synthesis. We also introduce in this chapter our representation of concurrent programs, which we use in the rest of this thesis.

Chapter 4 and Chapter 5 discuss two high-level property checking techniques.

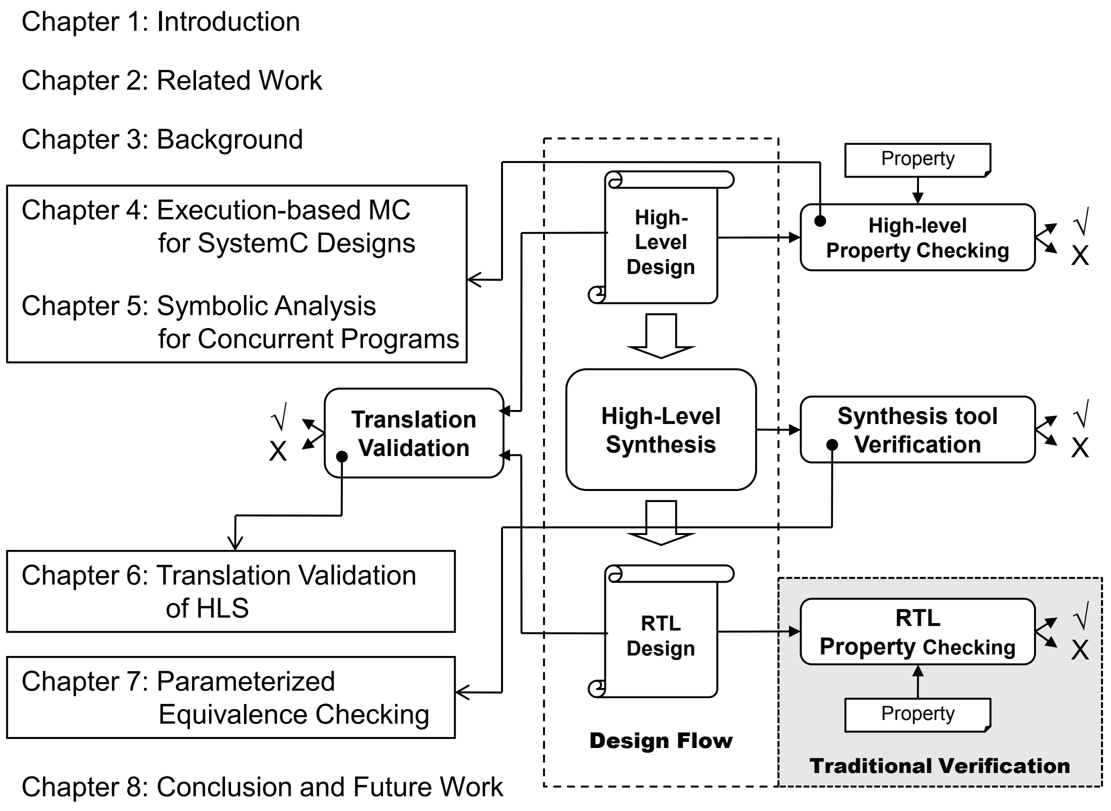

Chapter 1: Introduction
Chapter 2: Related Work
Chapter 3: Background
Chapter 4: Execution-based MC for SystemC Designs
Chapter 5: Symbolic Analysis for Concurrent Programs
Translation Validation
√
X
Chapter 6: Translation Validation of HLS
Chapter 7: Parameterized Equivalence Checking
Chapter 8: Conclusion and Future Work
High-Level Design
High-Level Synthesis
RTL Design
Design Flow
Property
High-level Property Checking
Synthesis tool Verification
RTL Property Checking
Property
Traditional Verification

Figure 1.2: Thesis organization

In Chapter 4, we present an execution-based model checking approach for the high-level language SystemC. In this approach, we start with a design written in SystemC, and then intelligently explore a subset of the possible executions till a certain depth to verify that the design satisfies a given property such as absence of deadlocks or assertion violations. Chapter 5, on the other hand discuss a symbolic analysis approach for concurrent C programs. We introduce the concept of MATs, which allows us to do partial-order reduction for symbolic algorithms. In both the chapters we present the details of our algorithm and experimental results.

Chapter 6 discusses a translation validation approach that proves the translation from high-level design to the scheduled design is correct. We describe in detail our algorithm that uses a bisimulation relation approach to automatically prove the equivalence between two concurrent programs. In this chapter, we also report our efforts to check the refinement of CSP programs and to validate the synthesis process of Spark, a parallelizing HLS framework.

In Chapter 7 we describe another approach that also proves the result of HLS process is correct. Our approach called *Parametrized Equivalence Checking* falls in the synthesis tool verification category. In this approach we generalize the translation validation technique of Chapter 6 to prove the optimizations performed by a HLS tool correct once and for all. We describe the details of our algorithm and experiments.

Finally, Chapter 8 wraps-up the thesis with a conclusion and a discussion of future work.

Chapter 2

Related Work

Each one of the three areas of high-level verification outlined in Chapter 1, namely high-level property checking, translation validation, and synthesis tool verification, have been explored in a wide variety of research efforts. In this chapter, we discuss various techniques from each of these areas that are directly relevant to our work.

2.1 High-Level Property Checking

The high-level designs written using languages like C, SystemC, SystemVerilog are mostly software programs with support for specialized hardware data types and other hardware features like synchronous concurrency, synchronization, and timing [GL97]. Thus, many efforts to use software verification tools to verify these designs have been explored. Model checking is the most prevalent automatic verification technique for software and hardware. It is a technique for verifying that a hardware or software system satisfy a given property (specification). These properties, which are usually expressed in temporal logic, typically encode deadlock and safety properties (e.g. assertion violations). In this section, we survey several software model checking techniques grouped as *explicit* and *symbolic* techniques.

2.1.1 Explicit Model Checking

In explicit state enumeration model checking, the reachable states of a design are generated using an exhaustive search algorithm. This technique explicitly stores the entire state space in memory and checks if certain error states are reachable. For finite state system this technique is both sound (i.e. whenever model checking cannot reach a given error state, it is guaranteed to not reach that error state ever in real execution) and complete (i.e. whenever model checking finds an error, it is guaranteed to be an error in real execution). However, as the size of the finite state spaces grow larger and larger, this technique suffers from the well known *state explosion* problem. To address the state explosion problem, researchers use techniques to construct the state space *on-the-fly* [Hol97] during the search, rather than generating all the states and transitions before the search. In addition, they use *bit-state hashing* [Hol97], in which the hash value of the reachable state is stored, instead of the state itself. Due to possible hash collision the bit-state hashing technique is unsound. Other techniques include *partial-order-reduction* [God95], *symmetry reduction* [ES96, CD93] and *compositional techniques* [CLM89].

Intuitively, the partial-order-reduction technique exploits the independence between parallel threads to compute a provably sufficient subset of the enabled transitions in each visited states such that if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations is guaranteed. Symmetry reduction on the other hand exploits symmetries in the program, and explores one element from each symmetry class. Compositional techniques decompose the original verification problem to related smaller problems such that the result of the original problem can be obtained by combining the smaller ones.

The most popular finite state explicit model checker for concurrent programs are SPIN [Hol97] and MURPHI [Dil96]. Both tools have been successfully used for verification of sequential circuits and protocols.

Moreover, in order to achieve scalability some systems give up completeness of the search and focus on the bug finding capabilities of model checking. For instance, one can bound the depth of the search and/or bound the number of context switches [MQ07]. This line of thought also leads to the execution-based model checking approach. These methods are typically used for improving the coverage of a test. Traditionally, in testing the user writes a test bench and runs it. Typically, the operating system scheduler executes only one fixed schedule out of the many possible behaviors. However, the scheduler of the execution based model checker systematically explore all possible behaviors of the program for a given test input and depth. The most striking benefit of this approach is the ease of implementing it, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this method is the idea of *stateless* [God97] search i.e., it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. Here again various reduction techniques like symmetry, partial-order [FG05], and abstraction have been explored.

Verisoft [God97] is the first tool in this domain, and it explores arbitrary code written in full fledged programming language like C or C++. It does so by modifying the OS scheduler and systematically exploring all possible interleavings. Java PathFinder [VHBP00] is a tool for Java programs that uses the virtual machine rather than OS scheduler to explore the different behaviors. CMC [MPC$^+$02] is another tool for C programs that improves the efficiency of the search by storing a hash of each visited state. Dynamic validation using execution-style model checking is also well adapted for validating SystemC designs [HMMCM06].

2.1.2 Symbolic Model Checking

The above reduction techniques like partial-order, address the state explosion problem for asynchronous concurrent systems (by reducing the number of interleavings that need to be explored). However, they are not so effective in the case of synchronous concurrent systems, which do not involve interleaving. Symbolic model checking techniques, on the other hand, are quite effective for both synchronous and asynchronous concurrent systems. Furthermore, the reduction techniques discussed in Section 2.1.1 including partial-order reduction are orthogonal and can be used in conjunction with symbolic techniques.

Symbolic algorithms manipulate sets of states, instead of individual states. These algorithms avoid ever building the complete state graph for the system; instead, they represent the graph implicitly using a formula in propositional logic. They can also represent infinite states using a single formula. For example the predicate $(x > 1 \wedge y > 1)$ denotes the set of all states in which the value of the variables x and y are both greater than 1. The first major step toward symbolic representation is the use of *Binary Decision Diagrams* (BDD) [Bry92]. BDDs are a canonical form representation for boolean formulas, and are particularly important for finite state programs, as these programs can be represented using boolean variables. BDDs are used in symbolic model checker like SMV [McM00] and have been instrumental in verifying hardware designs with very large state spaces [BCM$^+$90].

As in explicit model checking, one sometimes trades off completeness for bug finding capabilities of symbolic model checking. *Bounded Model Checking* (BMC) [BCC$^+$99] is one such algorithm that unroll the control flow graph (loop) for a fixed number of steps (say k), and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of Satisfiability (SAT) problem. This problem is then solved using a SAT [MMZ$^+$01] or SMT (Satisfiability Modulo Theory) [MB08] solver. BMC tools like CBMC [CKY03] and FSoft-BMC [IYG$^+$08] use iterative deepening depth-first search so that the above process can be repeated with larger and larger values of k until all possible violations have been ruled out.

Another area that has recently received lot of attention is *abstract model checking*, which trades off precision for efficiency. Abstraction [CC77, CC02] attempts to prove properties of a program by first simplifying it. Next, the reachability analysis is performed on the simplified (or abstract) domain, which usually satisfies some, but not all the properties of the original (or concrete) program. Generally, one requires the abstract domain and its semantics to be sound (i.e. the properties proved in the abstract semantics implies properties in the concrete semantics). However, typically, the abstraction is not complete (i.e. not all true properties in the concrete semantics are true in the abstract semantics). An example of abstraction is, to only consider boolean variables and the control flow of a program and ignore the values of non boolean variables. Although, such an abstraction may appear coarse, it is sometimes sufficient to prove properties like mutual exclusion.

The polyhedral abstract domain has been successfully used to check for array bounds violations [CH78]. Another interesting domain, *predicate abstraction* [GS97, BMMR01, DDP99, LBC05] is parameterized by a fixed finite set $\mathcal{B} = \{B_1, B_2, \cdots, B_k\}$ of first-order formulas (predicates) over the program variables, and consists of the lattice of Boolean formulas over $\mathcal{B}$ ordered by implication. A cube over $\mathcal{B}$ is a conjunction of possibly negated predicates from $\mathcal{B}$. The domain of predicate abstraction is the set of all cubes, and one cube is computed at each program point.

The goal of predicate abstraction is to compute a set of predicates from $\mathcal{B}$ at every program point. Thus, given a set of predicates $\mathcal{B}$, a program statement s, and a cube over $\mathcal{B}$ flowing into the statement, it computes the cube over $\mathcal{B}$ that flows out of the statement. For example, consider the set of predicates $\mathcal{B} = \{B_1, B_2\}$, where $B_1 \equiv (a = b)$ and $B_2 \equiv (a = b + 1)$. Given the cube $B_1 \wedge \neg B_2$ and the statement $a := b + 1$, then predicate abstraction would compute that the cube $\neg B_1 \wedge B_2$ should be propagated after the statement.

A problem with abstract model checking is that although the abstraction simulates the concrete program, when the abstraction does not satisfy a property, it does not mean that this property actually fails in the concrete program. When a property fails, the model checker produces a *counterexample*. A counterexample can be *genuine* i.e., can be reproduced on the concrete program, or *spurious* i.e., does not correspond to a real computation but arises due to the imprecision in the analysis. Counterexamples are checked against the real state space to make sure they are genuine. In the case when it is spurious, methods have been developed to automatically refine the abstract domain and get a more precise analysis which rules out the current counterexample and possibly many others, without losing soundness. This iterative strategy is called *Counter Example Guided Abstraction Refinement* (CEGAR).

SLAM [BMMR01] is a popular CEGAR based model checker for C programs. It was used successfully within Microsoft for device driver verification [BBC$^+$06] and has been developed into a commercial product (Static Driver Verifier, SDV). BLAST [BHJM07] is also a CEGAR based model checker that uses *lazy abstraction* [HJMS02]. The main idea of Blast is the observation that the computationally intensive steps of abstraction and refinement can be optimized by a tighter integration which would allow it to reuse the work performed in one iteration toward subsequent iterations. Lazy abstraction tightly couples abstraction and refinement by constructing the abstract model on-the-fly, and locally refining the model on-demand. MAGIC [CCG03] is another CEGAR based compositional model checking framework for concurrent C programs. Using MAGIC, the problem of verifying a large implementation can be naturally decomposed into the verification of a number of smaller, more manageable fragments. These fragments can be verified separately, enabling MAGIC to scale up to industrial size programs.

Advances in model checking and related techniques in the past several decades have allowed researchers to verify increasingly ambitious properties of software programs including device drivers, operating systems code, and large commercial applications. They have also enabled the verification of large hardware components like microprocessors. Although this is a significant step forward toward reducing the design-productivity-gap, state-of-the-art verification techniques are still far away from proving full correctness of programs.

2.2 Translation Validation

Once the design has been checked to satisfy certain properties using techniques discussed in Section 2.1, the next step is to make sure that those properties are preserved through the synthesis process. In this section we discuss a category of methods called translation validation which guarantee the preservation of safety properties through the synthesis process. Translation validation techniques are employed during synthesis to check that each transformation performed by the HLS tool preserves the semantics of the initial design. The initial design is called specification and the transformed design is called implementation. The validation step check for either *refinement* or *equivalence*. Typically, the implementation is said to be a refinement of the specification if the set of execution traces of the implementation is a subset of the set of execution traces of the specification. They are equivalent when the two sets are equal. In this section, we discuss different techniques for translation validation. Depending upon the core approach these techniques are primarily based on, they are divided into three categories: relational approach, model checking, and theorem proving.

2.2.1 Relational Approach

Relational approaches [KKM04, DBJ98, BDP00, KMS$^+$06] are used to check the correctness of the synthesis process by establishing a functional equivalence between the Control-Data Flow Graphs (CDFG) of the program, before and after each step of HLS. The equivalence is defined on some predefined *observable events* that are preserved across the transformations. Intuitively, the idea is to show that there exists a *simulation relation* R that matches a given program state in the implementation with the corresponding state in the specification. This simulation relation guarantees that for each execution sequence of observable events in the implementation, a related and equivalent execution sequence exists in the specification. The relation $R \subseteq State_1 \times State_2$ operates over the program states $State_1$ of the specification and the program states $State_2$ of the implementation. If $Start_1$ is the set of start states of the specification, $Start_2$ is the set of start states of the implementation, and $\sigma \rightarrow^e \sigma'$ denotes state σ stepping to state σ' with observable event

e, then the following conditions summarize the requirements for a correct refinement:

$$\begin{aligned}
&\forall \sigma_2 \in Start_2 \;.\; \exists \sigma_1 \in Start_1 \;.\; R(\sigma_1, \sigma_2) \\
&\forall \sigma_1 \in State_1, \sigma_2 \in State_2, \sigma_2' \in State_2 \;. \\
&\quad \sigma_2 \rightarrow^e \sigma_2' \wedge R(\sigma_1, \sigma_2) \Rightarrow \\
&\qquad \exists \sigma_1' \in State_1 \;.\; \sigma_1 \rightarrow^e \sigma_1' \wedge R(\sigma_1', \sigma_2')
\end{aligned}$$

These conditions respectively state that (1) for each starting state in the implementation, there must be a related state in the specification; and (2) if the specification and the implementation are in a pair of related states, and the implementation can proceed to produce observable events e, then the specification must also be able to proceed, producing the same events e, and the two resulting states must be related. The above conditions are the base case and the inductive case of a proof by induction showing that the implementation is a refinement of the specification.

One example of using the relational approach is Karfa et al.'s technique [KMS$^+$06] for establishing the equivalence between the initial Finite State Machine with Datapath (FSMD) and a scheduled FSMD. The technique introduces cut-points in the original and transformed FSMD automatons, which allows computations through the original and transformed FSMD to be seen as the concatenation of paths from cut-points to cut-points. The technique then establishes the equivalence by exploiting the structural similarities between related cut-points using weakest pre-condition.

Another example of the relational approach can be found in Dushina et al.'s proposed method [DBJ98] for checking the functional equivalence between a scheduled abstract FSM and the corresponding RTL after binding. The method establishes the equivalence transition by transition. In particular, for each transition in the RTL controller, it performs a symbolic execution of the associated RTL data path. The symbolic execution results are then syntactically compared with the data operations specified in the equivalent transition of the abstract FSM.

In general, relational approaches work well when the transformations preserve most of the program's control flow structure. Such transformations are called *structure-preserving* [ZPFG03] transformations. Unfortunately, relational approaches tend to be ineffective in the face of non structure-preserving transformations like loop unrolling, loop tiling and loop reordering. Despite these limitations, relational approaches are very useful in practice: with only a fraction of the development cost of an HLS tool, they can uncover bugs that elude testing.

2.2.2 Model Checking

Techniques involving model checking [ABRM98, Bla00] are used for verifying register-transfer logic against its scheduled behavior. The key idea is to partition the equivalence checking task into two simpler subtasks, verifying the validity of register sharing/binding, and verifying correct synthesis of the RTL interconnect and control. The success of these methods can be attributed to the observation that the state space explosion in most designs is caused by the data path registers rather than the number of control states.

The following algorithm outlines the method of verifying the validity of register sharing.

- Identify paths in the scheduled graph along which potential conflicts can occur. During this step no interpretation of the data path is done. If no conflict is identified then verification successfully terminates.

- Otherwise for each violation the set of all conflict paths is summarized in a reduced Conflict Sub-Graph (CSG).

- The reduced CSG is then checked to find out if the conflict was benign. If a conflict is detected during the checking, then a logically possible path with incorrect register binding has been detected. In this case the appropriate path is shown to the user as a counterexample.

- Else, if it ends without any conflict detected, all possible conflict paths are logically impossible and the verification algorithm successfully terminates.

Ashar et al. [ABRM98] analyzed potential conflicts by means of structural methods, and then the reduced CSG is checked for satisfiability by the VIS model checker [TVGSV95]. Whereas Blank [Bla00] identifies possible conflicts using a symbolic model checker [BCM$^+$90]. The result of the analysis is summarized in a reduced internal representation called Language of Labeled Segments (LLS) [Hin98], which is then checked by symbolic simulation [Moo98]. However, symbolic simulation allows reasoning for a defined finite number of steps. Thus, loops in the program cannot be verified for an arbitrary number of iterations.

Ashar et al. also presented an algorithm to verify the correct synthesis of the RTL interconnect and control [ABRM98]. This part of equivalence checking is done state-by-state, i.e., for each state in the schedule, the computations performed in that state are shown to be equivalent to those performed in the RTL implementation for the same state. The equivalence is shown using symbolic simulation.

2.2.3 Theorem Proving

Although most of the translation validation approaches discussed so far use theorem provers in some way, the theorem prover is not at the center of the approach. The Correctness Condition Generator [MV00], on the other hand, is primarily based on a theorem proving technique. This approach assumes that the synthesis tool can identify the binding relation between specification variables and registers in the RTL design, and between the states in the behavior and the corresponding states in the RTL design. A correctness condition generator is tightly integrated with the high-level synthesis tool to automatically generate (1) formal specifications of the behavior and the RTL design including the data path and the controller, (2) the correctness lemmas that establish equivalence between the synthesized RTL design and its behavioral specification, and (3) proof scripts for these lemmas that can be submitted to a higher-order logic proof checker without further human interaction. The tight integration of the synthesis process with the theorem prover allows the theorem prover to gather information about what kinds of transformations were performed, and therefore better reason about them.

2.3 Synthesis Tool Verification

Another attractive way of proving that an HLS tool produces correct RTL is to verify the correctness of HLS tool once and for all, before it is ever run once.

One can categorize such techniques into three broad classes: (1) *formal assertions*, which can be used to guarantee the correctness of the synthesis tool, (2) *transformational synthesis tools*, which are correct by construction, and (3) *witness generators*, which recreate the steps that an existing HLS tool has performed using formally verified transformations.

2.3.1 Formal Assertions

Narasimhan et al. proposed a Formal Assertions approach [NTR$^+$01, Nar98, NV98] to building a verified high-level synthesis system, called *Asserta*. The approach works under the following premise: If each stage in the system, like scheduling, register optimization, interconnect optimization etc. can be verified to perform correct transformations on the input specification, then by compositionality, we can assert that the resulting RTL design is equivalent to its input specification. This technique has the following four main steps.

1. *Characterization*: A base specification model is identified for each synthesis task. The base specification model is usually a tight set of correctness properties that completely characterizes the synthesis task.

2. *Formalization*: The base specification model is then formalized as a collection of theorems in a higher order logic theorem proving environment, which form the base formal assertions. An algorithm is also chosen to realize the corresponding synthesis task and is described in the same formal environment.

3. *Verification*: The formal description of the algorithm is verified against the base theorems. Inconsistencies in the base model are identified during the verification exercise. Furthermore, the model is enhanced with several additional formal assertions derived during verification. The formal assertions now represent the invariants in the algorithm.

4. *Formal Assertions Embedding*: In the next step a software implementation of the algorithm that was formally verified in the previous stage is developed. The much enhanced and formally verified set of formal assertions is then embedded within this software implementation as program assertions.

During synthesis, the implementation of each task is continually evaluated against its specification model specified by these assertions and any design error during synthesis can be detected.

Asserta [Nar98] is a high-level synthesis system developed to show the effectiveness of assertion-based verification techniques to generating first-time correct RTL designs. The synthesis engine has three main stages, namely scheduling, register optimization and interconnect generation. The proof effort was conducted using Prototype Verification System (PVS) [ORS92], a higher order logic theorem prover.

Since the main tasks of Asserta have been verified, it can be used with an increased degree of confidence. This approach is also not affected by the state space or complexities of any synthesized RTL design. However, the correctness of the system depends on the completeness and correctness of the base assertions. Another concern is that during the formal assertions embedding step, due to difference in the expressive power of logic and software program, the translation process often could get quite complicated and finally, the correctness of the method hinges on this translation process. It is also hard to generate a tight base specification for all the steps of the synthesis process. Thus, although Asserta is a first step toward achieving correct synthesis, verifying large synthesis programs is quite tedious and complex.

2.3.2 Transformational Synthesis Tools

The basic idea of this method is to determine a set of transformations, which when applied to an initial specification, transform the source into the required implementation. This transformations are then embedded in a theorem prover to prove their correctness. The correctness of the HLS system thus follows from a 'correct by construction' argument.

Transformational synthesis is an area that has been widely explored [SR97, JB91, HDL89, Lar96, EBK96, Raj95]. Various tools have been developed in the recent past, which mainly differ in the expressiveness of their input language, the theorem prover used and the type of transformations allowed. Sharp et al. [SR97] developed the T-Ruby design system, where the Ruby language is used for specifying VLSI circuits and the theorem prover Isabelle [Pau94] is used to formalize the correctness-preserving transformations. DDD [JB91] is another system, which is a based on functional algebra. Both systems uses hardware specific calculus to describe a design. The following are few systems based on behavioral transformations. Veritas [HDL89] is a theorem prover based on an extension of typed higher order logic, which provides an interactive environment for converting the specification into an implementation. Larsson [Lar96] presented a transformational approach to digital system design based on the HOL proof system [Gor88]. Hash [EBK96] is another system based on the theorem prover HOL [Gor88]. McFarland [McF93b] investigated the correctness of behavioral transformations using behavior expressions. Rajan [Raj95] on the other hand, used the PVS [ORS92] theorem proving system to specify and verify behavioral transformations.

However, unlike the formal assertion technique presented in Section 2.3.1, techniques based on transformational synthesis reason only about the specification of the

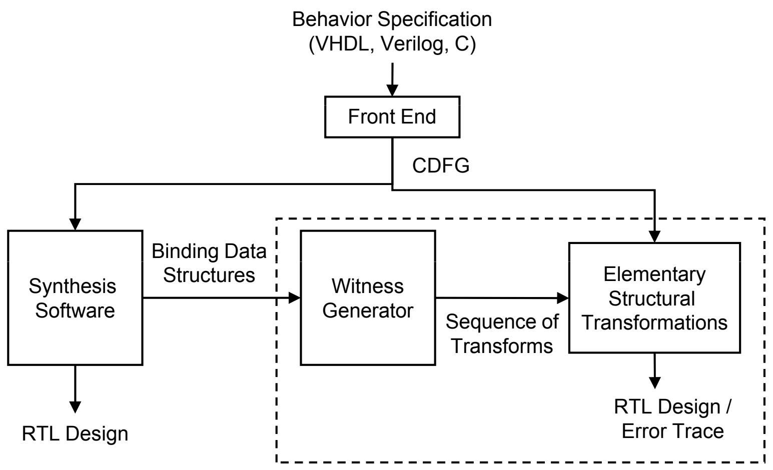


Figure 2.1: Using a witness generator system to validate synthesis tools

transformations, not their software implementations, which is where many of the bugs arise.

2.3.3 Witness Generator

The main idea behind witness generator techniques [RTV00, MHMP02, EHR99] is to use a set of behavior-preserving elementary transformations for validating an existing non transformational synthesis system by discovering and to some extent isolating software errors.

Figure 2.1 shows an overview of the witness generator approach. The source program is first converted into a CDFG, after which point the CDFG passes through a regular unverified HLS process. Following the regular HLS process, the CDFG is also passed to a transformational system that consists of a set of elementary structural transformations, all of which have been formally verified given a set of preconditions. These transformations are sequenced together by the witness generator, whose goal is to find a sequence of elementary transformations which when applied to the initial design, achieve the same RTL outcome. For this technique to be broadly applicable, the set of elementary transformations must collectively capture a wide variety of synthesis algorithms.

The task of the witness generator is facilitated by the following information, which is provided by the synthesis tool:

- The outcome of a synthesis task can be captured by a simple data structure (binding data structure) such that any algorithm for this task can record its outcome in this data structure. For example, the outcome of any scheduling algorithm can be recorded as a schedule table which records the mapping between operations to control steps and the outcome of any register allocation algorithm can be recorded as mapping from variables to registers.

- It is possible to generate a sequence of elementary transformations to perform the same task by examining this data structure, without any knowledge of the synthesis algorithm used to perform the task.

When a precondition fails during the execution of the sequence of transformations identified by the witness generator, the sequence applied so far forms a counter-example that can be presented to the user.

Radhakrishnan et al. [RTV00] identified a set of six elementary transformations which were sufficient to emulate the effect of many existing high-level synthesis algorithms. Each of these transformations is mechanically proved in PVS [ORS92] to preserve the computational behavior.

Eveking et al. [EHR99] uses a similar approach to verify the correctness of various scheduling algorithms. They represented the initial CDFG using the LLS [Hin98] internal language. After that, the process of equivalence verification consists of a number of computationally equivalent LLS transformation steps which *assimilate* the original design to the scheduled design.

Recently, Mendías et al. [MHMP02] used equational specification to describe behaviors and/or structures, in a elaborate formal framework called *Fresh*. In Fresh, seven formal derivation rules, classified into structural and behavioral were used to transform the initial design to the RTL design.

The above systems essentially recreate, within a formal framework, each of the design decisions taken by an external (and potentially incorrect) HLS algorithm. The latest HLS tools are complex and use a variety of transformations to optimize the synthesis result for metrics like area, performance and power. As a result, it is becoming increasingly difficult to find a small set of correct transformations that can recreate all the design decisions taken by external HLS tools.

2.4 Summary

In this chapter we discussed various state-of-the-art related work in the area of high-level verification. The last decade witnessed great improvements in formal methods and HLS. Recently, many commercial formal verification tools for system-level designs such as Hector [KBP07], SLEC [Sys], SCADE Design Verifier [Tec], and Statemate [Cor] have become available. However, their adoption is in the early stages and the tools are often limited in the quality of the results and the kinds of correctness guarantees that are provided. In the following chapters we will discuss the high-level verification techniques that we developed. Moreover, in each of the following chapters, we again discuss some more related works specific to the chapter and compare them with our approach.

Acknowledgements: Chapter 2, in part, have been accepted for publication as “High-Level Verification” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *IPSJ Transactions on System LSI Design Methodology* [KLG09]. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Background

We envision a design methodology that is built around advances in high-level design and verification to improve the quality and time to design microelectronic systems. In this chapter, we will present a brief overview of the three different parts of high-level verification as shown in Figure 1.1 on which our algorithms are applied. We first present in Section 3.1 and in Section 3.2 a description of high-level designs and RTL designs respectively. We then in Section 3.3 give a brief introduction of high-level synthesis. Finally, in Section 3.4 we introduce our program representation scheme that is used throughout the dissertation.

3.1 High-Level Design

A high-level design is a behavioral or algorithmic specification of a system. This specification typically is written in a high-level language such as behavioral VHDL, C and C++ (and variants). The main reason to use a high-level language is to be able to describe both the hardware and software components of a design, and to allow large system designs to be modeled uniformly. The enormous flexibility in describing computations in a high-level language enables a designer to capture design description at multiple levels of abstraction from behavioral descriptions to transaction level modeling (TLM) [Swa06, GLMS02, CG03]. Intuitively, high-level design gives an abstract view of the system. When describing behaviors in a high-level language, designers often use programming constructs such as conditionals and loops for programming convenience often with no notion of how these constructs may affect synthesis results. For example, SystemC [GLMS02] TLM supports new synchronization procedures such as wait-notify, which makes current techniques for synthesis inapplicable. One of the key aspects of high-level design is the ability to provide a golden reference of the system in an early phase of the development process.

In this thesis, we use various high-level languages as input for our techniques. In Chapter 4 we will discuss a property checking approach for the SystemC language. In Chapter 5 we describe another approach that works on a concurrent C program. For our translation validation approach described in Chapter 6, we use two high-level languages namely, C and CSP.

3.2 RTL Design

At the current state of practice, RTL designs are generally considered low-level designs consisting of structural implementation details. The RTL describes the exact behavior of the digital circuits on the chip, as well as the interconnections to inputs and outputs. The structural implementation usually consists of a data path, a controller and memory elements. Figure 3.2 shows the controller and data path for a RTL design. The data path consists of component instances such as ALU, multiplier, registers, and multiplexers selected from a RTL component library. The controller is a finite-state machine (FSM) describing an ordering of the operations in the data path. A tiny functional error in the RTL design can sometimes make the entire chip inoperable. Furthermore, writing RTL designs are often tedious, complex, and error-prone as such it is desirable to have a good high level design and then use incremental refinement process to generate the final RTL.

In this thesis, we do not consider RTL property verification. However, for the purpose of translation validation we want to check the equivalence between a pair of high-level design and RTL design. Unfortunately, checking equivalence between the high-level design and RTL design is a hard problem. As such in Chapter 6 we describe an approach that validates the equivalence between a high-level design written in C and the scheduled design (described in the next section), which is an intermediate low-level design.

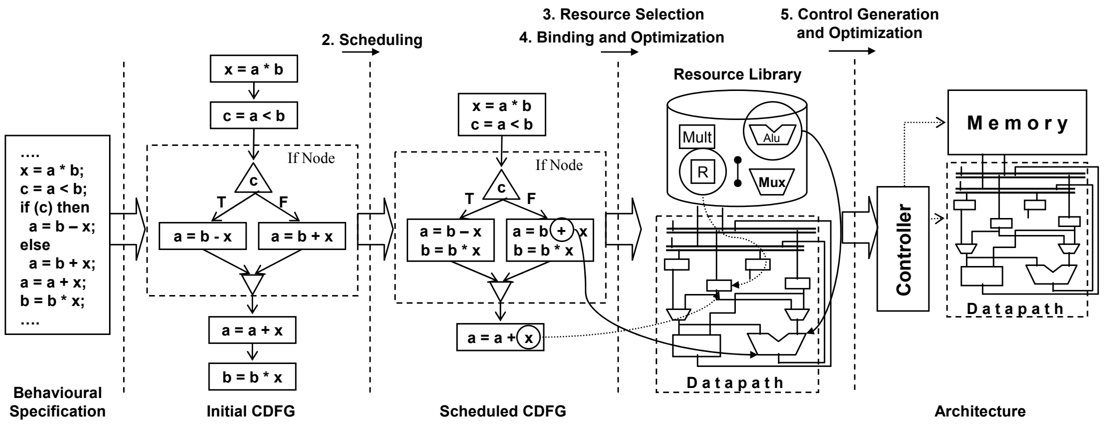


Figure 3.1: Stepwise transformation during high-level synthesis process

3.3 High-Level Synthesis

HLS can be seen as stepwise transformation of a high-level design into a RTL design as shown in Figure 3.1. Figure 3.2 shows the RTL design for the example in Figure 3.1. Different HLS tool produces different RTL design for the same high-level input as it minimizes various metrics like area, power and timing. HLS starts by capturing the behavioral description in an intermediate representation, usually a control data flow graph (CDFG). Thereafter the HLS problem has usually been solved by dividing the problem into several sub-tasks [GDGN03]. Typically the sub-tasks are:

1. **Allocation:** This task consists of determining the number of resources that need to be allocated to synthesize the hardware circuit (not shown in the Figure). Typically, designers can specify an allocation in terms of the number of resources of each resource type. Resources consist of functional units (like adders and multipliers), registers and interconnection components (such as multiplexers and buses). The allocated resources constitute the *resource library* for the design.

   In our example (Figure 3.1 and Figure 3.2), the resource library contains a multiplier, an ALU, 2 multiplexers, 4 registers, 2 buses and a memory component (not all components are shown in the Figure). Usually, a designer chooses these components based on several design constraints like area, performance and power.

2. **Scheduling:** This step determines the time step or the clock cycle in which each operation of the design executes. The ordering between the “scheduled” opera-

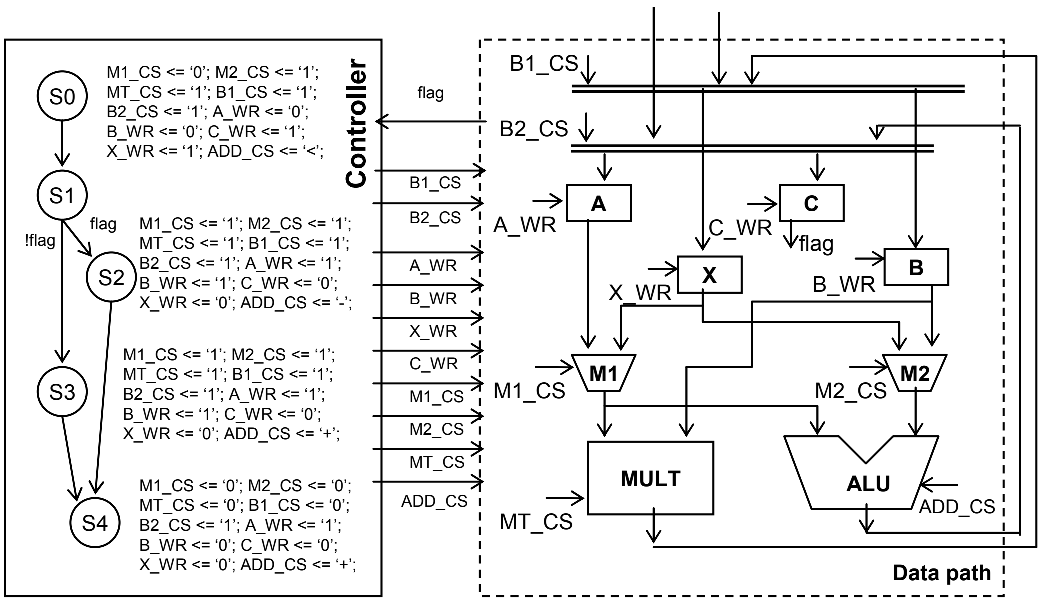

Controller
S0
M1_CS <= '0'; M2_CS <= '1'; MT_CS <= '1'; B1_CS <= '1'; B2_CS <= '1'; A_WR <= '0'; B_WR <= '0'; C_WR <= '1'; X_WR <= '1'; ADD_CS <= '<';
S1
flag
!flag
S2
M1_CS <= '1'; M2_CS <= '1'; MT_CS <= '1'; B1_CS <= '1'; B2_CS <= '1'; A_WR <= '1'; B_WR <= '1'; C_WR <= '0'; X_WR <= '0'; ADD_CS <= '-';
S3
M1_CS <= '1'; M2_CS <= '1'; MT_CS <= '1'; B1_CS <= '1'; B2_CS <= '1'; A_WR <= '1'; B_WR <= '1'; C_WR <= '0'; X_WR <= '0'; ADD_CS <= '+';
S4
M1_CS <= '0'; M2_CS <= '0'; MT_CS <= '0'; B1_CS <= '0'; B2_CS <= '1'; A_WR <= '1'; B_WR <= '0'; C_WR <= '0'; X_WR <= '0'; ADD_CS <= '+';
flag
B1_CS
B2_CS
A_WR
B_WR
X_WR
C_WR
M1_CS
M2_CS
MT_CS
ADD_CS
B1_CS
B2_CS
A
A_WR
C
C_WR
flag
X
X_WR
B
B_WR
M1
M1_CS
M2
M2_CS
MULT
MT_CS
ALU
ADD_CS
Data path

Figure 3.2: The controller and the data path for the example in Figure 3.1

tions is constrained by the data (and possibly control) dependencies between the operations. Scheduling is often done under constraints on the number of resources.

Resource constraint scheduling algorithms are highly dependent on the resource allocation task. For example, since in the allocation step the designer chose a multiplier and an ALU, we were able to schedule both the statements ($x = a * b$ and $c = a < b$) in one cycle (C_1). In contrast, if only an ALU is chosen then both the operations have to be scheduled in different cycles.

Schedulers also do code motions to enhance *concurrency* and hence improving resource utilization. In our example, the statement $b = b * x$ has been moved inside the 'if block' by the scheduler to schedule it in cycle C_2 instead of cycle C_3.

3. **Resource selection:** This task of determines the resource type from the resource library that an operation executes on. The need for this task arises because there are several resources of different types (and different area and timing) that an operation may execute on. For example, an addition may execute on an adder, an ALU, or a multiply-accumulate unit. Resource selection must make a judicious choice between different resources such that a metric like area or timing is minimized.

   In the example, all multiplication operations are done using the multiplier and the addition and comparison operations are executed in the ALU (see Figure 3.1 and 3.2).

4. **Binding and Optimization:** This task determines the mapping between the operations, variables and data (and control) transfers in the design graph and the specific resources in the resource library. Hence, operations are mapped to functional units, variables to registers and data/control transfers to interconnection components. Optimizations deals with the minimization of the physical components in the synthesized design.

   In our case, the variables $a, b, c, and\ x$ in the behavioral description are bound to the registers $A, B, C, and\ X$ respectively. Figure 3.1 shows the binding of the variable a to the register A and the operation $+$ to the ALU. Notice that multiple operations can be bound to the same resource. The complete binding of the operations and the data/control transfers to functional units and interconnection components are shown in Figure 3.2.

5. **Control Generation and Optimization:** Control synthesis generates a control unit (usually FSM) that implements the schedule. This control unit generates control signals that control the flow of data through the data path (i.e. through the multiplexers). Control optimization deals with minimizing metrics such as area and power.

   Operations in the scheduled CDFG are replaced by the concrete values of control signals going to the data path. Thus, for example the concurrent operations ($x = a * b\ and\ c = a < b$) are replaced by the following signals.

```
M1_CS <= '0',    M2_CS <= '1',   MT_CS <= '1',
B1_CS <= '1',    B2_CS <= '1',   A_WR <= '0',
B_WR <= '0',     C_WR <= '1',    X_WR <= '1',
ADD_CS <= '<'
```

   According to these control signals, the contents of the registers A and B are simultaneously fed as inputs to the $MULT$ and the ALU units and then the result of $MULT$ and ALU is saved in the registers X and C respectively.

HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [WC91, Lin97, GDGN03]. These

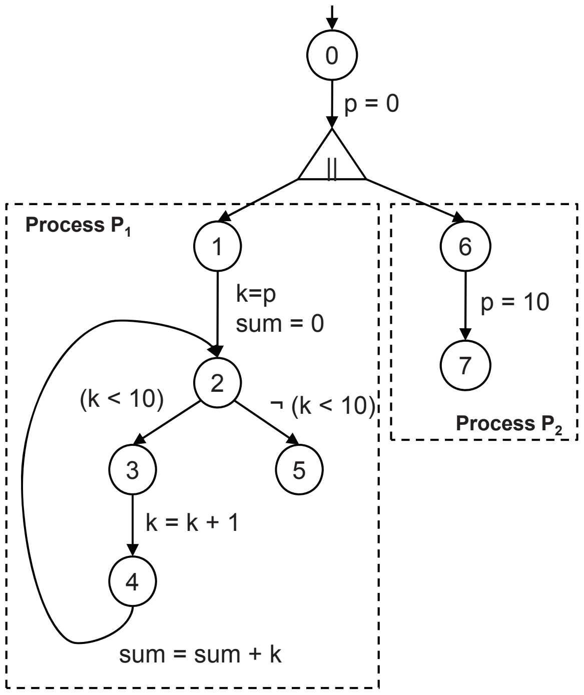


Figure 3.3: Our Concurrent Control Flow Graph (CCFG) representation

tools are usually very large and complex piece of software, as such their implementation are prone to errors. In Chapter 6 we discuss an approach that validates parts of a parallelizing HLS tool called Spark [GDGN03]. Furthermore, in Chapter 7 we describe a technique to once-and-for-all prove the correctness of important parts of HLS tools.

3.4 Representation of Concurrent Programs

We visually represent a concurrent program using an internal Concurrent Control Flow Graph (CCFG) representation. Concurrent behavior of programs can be modeled as synchronous or asynchronous (described in Section 5.1). The CCFG of a simple example is shown in Figure 3.3. In general, we omit the details of the actual code, because the CCFG representation is complete. Our example consist of two processes P_1 and P_2. We use a node with the symbol || to denote the asynchronous parallel composition of child processes. This example computes the sum from 1 to 10 in the variable sum if the process P_1 is executed before P_2, otherwise sum is 0.

We represent the behavior of a concurrent program using a state transition system or a transition diagram (defined below), depending on if our technique works over program states or over control and data states separately. Semantically a concurrent program in our case consists of a set of processes (threads). The control structure of a program is represented in terms of *generalized program locations*. A generalized program location represents a point of control in the program (possibly concurrent). A generalized program location can either be a node identifier, or it can be a pair of two generalized program locations, representing the state of two processes running in parallel. For instance, a generalized program location for the example in Figure 3.3 is $(2, 6)$, which means the control of the program is at location 2 in P_1 and at location 6 in P_2.

Let $\mathcal{L}$ denote the finite set of generalized program locations, VAR denote the set of variables and VAL denote the domain of values. Next we define the two representations.

3.4.1 State Transition System

In this representation, we define a program state to be a function $VAR \times \mathcal{L} \rightarrow VAL$, assigning values to variables and program locations. A transition then describes how the system moves from one state to a subsequent state. Let $\mathcal{T}$ denote the set of all transitions of the system and Σ the set of all program states. An i^{th} transition of process P is denoted by P^i. For $t = P^i \in \mathcal{T}$ we denote, $\mathsf{Process}(t)$ as the process P.

Definition 1 (Runnable). *A transition $t \in \mathcal{T}$ is runnable in state $\sigma \in \Sigma$, written $t \in runnable(\sigma)$ if it can be executed in σ.*

If $t \in runnable(\sigma)$, then we say the execution of t from σ produces a successor state σ', written $\sigma \xrightarrow{t} \sigma'$. We write $\sigma \overset{w}{\Longrightarrow} \sigma'$ to mean that the execution of the finite sequence $w \in \mathcal{T}^*$ leads from σ to σ'. A state σ, where $runnable(\sigma) = \emptyset$ is called a deadlock, or a terminating state. Formally a state transition system is defined as follows.

Definition 2 (State Transition System). *A state transition system is a tuple $M = (\Sigma, \sigma_0, \Delta)$, where σ_0 is the initial state of the system and $\Delta \subseteq \Sigma \times \Sigma$ is the transition relation defined by*

$$(\sigma, \sigma') \in \Delta \quad \textit{iff} \quad \exists\, t \in \mathcal{T} : \sigma \xrightarrow{t} \sigma'$$

Definition 3 (Co-Runnable). *Two transitions $t_1, t_2 \in \mathcal{T}$ are* co-runnable*, written $\mathsf{CoRunnable}(t_1, t_2)$ if $\exists\, \sigma \in \Sigma$ such that both $t_1, t_2 \in runnable(\sigma)$.*

An execution of the program is defined by a trace of the system.

Definition 4 (Trace)**.** *A trace $\phi \in \mathcal{T}^*$ of M is a finite (possibly empty) sequence of transitions $t_0, \cdots, t_{n-1}$ where there exists states $\sigma_0, \cdots \sigma_n$ such that σ_0 is the initial state of M and $\sigma_0 \xrightarrow{t_0} \sigma_1 \cdots \xrightarrow{t_{n-1}} \sigma_n$.*

For a given trace $\phi = t_0, \cdots, t_n$; ϕ_i represents the transition t_i; $\phi_{0\cdots i}$ denotes the trace $t_0, \cdots, t_i$; $\mathsf{Pre}(\phi, i)$ denotes the state σ_i and $\mathsf{Post}(\phi, i)$ denotes the state σ_{i+1}.

3.4.2 Transition Diagram

In this representation, the program states are described using separate control states and data states. The program is described in terms of *generalized program locations* and *program transitions*. A transition describes how the data state changes from one program location to another. We represent these transitions by *instructions*.

More formally, we define a data state to be a function $VAR \rightarrow VAL$ assigning values to variables. We denote by Σ the set of all data states. Note that we make some notation abuse here for ease of readability. We use Σ for program states in state transition system and again Σ for data states in transition diagram. Also when there is no ambiguity, we use *state* to mean *data state* in the context of transition diagram. We define an instruction to be a pair (c, f) where $c : \Sigma \rightarrow \mathcal{B}$ is a predicate and $f : \Sigma \rightarrow \Sigma$ is a state transformation function. The predicate c is the condition under which the state transformation function f can happen. For instance, in Figure 3.3 the instruction on the edge $(2, 3)$ has $c(\sigma) = (\sigma(\mathsf{k}) < 10)$ and $f(\sigma) = \sigma$, whereas the instruction on the edge (3, 4) has $c(\sigma) = \mathit{true}$ and $f(\sigma) = \sigma[\mathsf{k} \mapsto (\sigma(\mathsf{k}) + 1)]$.

Finally a transition diagram is defined as follows.

Definition 5 (Transition Diagram)**.** *A transition diagram π is a tuple $(\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, where $\mathcal{I}$ is a finite set of instructions, $\rightarrow \subseteq \mathcal{L} \times \mathcal{I} \times \mathcal{L}$ is a finite set of triples (gl, i, gl') called transitions, and $\iota \in \mathcal{L}$ is the entry location. We write $gl \xrightarrow{i} gl'$ to denote $(gl, i, gl') \in \rightarrow$. We use ϵ to represent the exit location of π.*

Definition 6 (Configuration)**.** *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, we define a configuration to be a pair $\langle gl, \sigma \rangle$, where $gl \in \mathcal{L}$ and $\sigma \in \Sigma$.*

Definition 7 (Semantic Step)**.** *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, two configurations $\langle gl, \sigma \rangle$ and $\langle gl', \sigma' \rangle$, and an instruction $i \in \mathcal{I}$, the semantic step relation is defined as follows:*

$$\langle gl, \sigma \rangle \overset{i}{\rightsquigarrow} \langle gl', \sigma' \rangle \quad \mathit{iff} \quad gl \xrightarrow{i} gl' \ \wedge\ i = (c, f) \ \wedge\ c(\sigma) = \mathit{true} \ \wedge\ \sigma' = f(\sigma).$$

Definition 8 (Execution Sequence)**.** *For a given transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, an execution sequence η starting in configuration $\langle gl_0, \sigma_0 \rangle$ is a sequence of configurations such that:*

$$\langle gl_0, \sigma_0 \rangle \overset{i_1}{\rightsquigarrow} \langle gl_1, \sigma_1 \rangle \overset{i_2}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n \rangle$$

We denote by $\mathcal{N}$ the set of all execution sequences. We use the shorthand notation $\eta\langle \pi, gl_0, \sigma_0 \rangle$ to represent an execution sequence η starting in configuration $\langle gl_0, \sigma_0 \rangle$ in π.

3.5 Summary

In this chapter, we presented a brief overview of the three main concepts related to our formulation of high-level verification namely, high-level design, RTL design, and HLS. We first presented a description of high-level design and RTL design in Section 3.1 and Section 3.2. We also mentioned the different high-level languages and RTL representation that we use in this thesis. We then briefly discussed in Section 3.3 the various steps of HLS using a simple example. Finally, in Section 3.4, we introduced the two kind of concurrent program representation that we use throughout this dissertation. In particular, we defined *state transition system* and *transition diagram* representation and related definitions. We use state transition system if our technique works over program states, and we use transition diagram if our technique works over control and data states separately.

Chapter 4

Execution-based Model Checking for High-Level Designs

In this chapter, we present an high-level property checking approach. We begin with a general description of verification of concurrent programs, and then describe it for a high-level language called SystemC [GLMS02]. In this approach, we start with a design written in SystemC, and then use *model checking* techniques to verify that the design satisfies a given property such as the absence of deadlocks or assertion violations.

4.1 Verification of Concurrent Programs

Verification of multi-threaded concurrent programs is hard due to complex and unexpected interleaving between the threads. In general, the problem of verifying two-threaded programs (with unbounded stacks) is undecidable [Ram00]. Therefore for practical reasons, the verification techniques often trades off completeness or precision or sometimes both, to address the scalability of the problem, thereby focusing only on their bug finding capabilities. The verification model is typically obtained by composing individual thread models using interleaving semantics, and model checkers are applied to systematically explore the global state space. Due to the potentially large number of interleavings of transitions from different threads, the global state space can be, in the worst case, the product state space of individual thread state space. To combat the state explosion problem, most methods employ partial-order reduction (POR) techniques to restrict the state-traversal to only a representative subset of all interleavings, thereby, avoiding exploring the redundant interleaving among independent transitions [God95]. Explicit model checkers [Hol97, God97, Dil96, FG05] explore the states and transitions of concurrent system by explicit enumeration, while symbolic model checkers [ABH$^+$01, KGS06, RG05, WYKG08, GG08] manipulate sets of states, instead of individual states. Symbolic algorithms avoid explicitly building the complete state graph for the system; instead, they represent the state space implicitly using a formula in decidable subset of first-order logic. This formula is then solved using a SAT or SMT solver. In this chapter, we discuss an explicit model checking technique for SystemC designs, and in the next chapter we focus on a symbolic approach based on SMT.

4.2 Overview of SystemC

SystemC is a system description language that enables a designer to write designs at various levels of abstraction. These are particularly useful in behavioral/algorithmic and transaction level modeling (TLM) [Swa06, GLMS02, CG03]. The idea of SystemC TLM is to provide a golden reference of the system in an early phase of the development process and allow fast simulation. This design abstraction supports new synchronization procedures such as wait-notify, which make current techniques for RTL validation inapplicable. SystemC is a set of library routines and macros implemented in C++, which makes it possible to simulate concurrent processes, each described by ordinary C++ syntax. Instantiated in the SystemC framework, the processes described in this manner may communicate in a simulated real-time environment, using shared variables, events and signals. SystemC is both a description language and a simulation kernel. The code written will compile together with the library's simulation kernel to give an executable that behaves like the described model when it is run. In Section 4.5 and Section 4.6 we will discuss some more language features of SystemC.

4.3 Problem Statement

Simulation has so far been the "workhorse" for validating SystemC designs. As pointed out in [Var07], adapting software formal verification techniques to SystemC has been formidable task, mainly due to its object-oriented nature and its support for both synchronous and asynchronous semantics of concurrency along with a notion of time. Furthermore, SystemC allows features such as co-operative multitasking, delayed/immediate notification, wait-to-wait atomicity, blocking and non-blocking variable updates. In the absence of accepted formal semantics, SystemC models and methods attempt to speed up simulation. However, simulation can not guarantee completeness without being exhaustive, hence the need for formal verification techniques to improve system level simulation coverage.

4.4 Overview of Our Approach

Our goal is to devise methods to check all possible execution traces of a SystemC description. We focus on using formal verification techniques developed for software to extend dynamic validation of SystemC TLM designs. We use an execution-based model checking approach, which for a given test input and depth, systematically explores all possible behaviors of the design (due to asynchronous concurrency). The most striking benefit of this approach is that it can analyze feature-rich programming languages like C++, as it sidesteps the need to formally represent the semantics of the programming language as a transition relation. Another key aspect of this approach is the idea of *stateless* [God97] search, meaning it stores no state representations in memory but only information about which transitions have been executed so far. Although stateless search reduces the storage requirements, a significant challenge for this approach is how to handle the exponential number of paths in the program. In what follows, we assume the representative inputs are already provided, possibly using techniques presented in [GED07] and the execution terminates. Thus, we focus our discussion mainly on detecting deadlocks, write-conflicts and safety property violations such as assertion violations. Note that termination can be guaranteed in SystemC by bounding the execution length during simulation.

To cope with the exponential number of paths, we use a combination of static and dynamic POR techniques. In particular, we first use static analysis techniques to compute if two atomic blocks are *independent*, meaning that their execution does not interfere with each other, and changing their order of execution will not alter their combined effect. Next, we start by executing one random trace of the program until completion, and then dynamically compute backtracking points along the trace that identify alternative transitions that need to be explored because they may lead to different final states. However, unlike dynamic techniques [HMMCM06, FG05] we use the information obtained by static analysis in a *query-based* approach, rather than dynamically collecting the information and analyzing it during runtime. Using static information we trade off precision for performance. We chose performance since for most SystemC designs we can find the dependency relation quite precisely by using static analysis only. Intuitively, our approach infers the persistent sets dynamically using information obtained by static analysis. To further reduce the number of explored traces we use the *sleep sets* in conjunction with the above technique.

Moreover, we adapt the POR techniques by using SystemC specific semantics to further improve the efficiency of the algorithms. Adaptations are needed because in SystemC: processes are co-operatively multitasking; supports the concept of δ-cycle, which reduces the analysis of backtracking points immensely; supports signal variables that do not change values until an update phase; synchronization is done using events instead of locks; and enabled processes cannot be disabled by another one.

Contributions

The main contributions of our approach are:

1. We propose a novel *query-based* approach that combines static and dynamic POR techniques to cover all possible executions of a SystemC design. We reduce the runtime overhead by computing the dependency information statically, and use it during runtime, without much loss of precision.

2. We use SystemC specific semantics to further improve the efficiency of the POR techniques. In SystemC, processes are co-operatively multitasking and supports the concept of δ-cycle. This synchronous semantics of SystemC reduces the size of persistent set and consequently reduces the analysis of backtracking points immensely.

3. We use the Open SystemC Initiative's (OSCI) SystemC simulator [Ini05] to implement our algorithm of exploring all possible behaviors of a SystemC design in a validating system called Satya. We use Satya to check the correctness of a variety of small examples and two benchmark designs. In particular, we were able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of OSCI repository), which may not have been found by simulation. We also applied our tool on an industrial benchmark namely the TAC platform [Mic05].

```
1.  int MAX, $num = 0,\ i = 0$
2.  char $data[2]$
3.  sc_event $e$
4.  $Boolean\ timer = false$

5.  **process** P$_1$()
6.      **while** $(i <$ MAX$)$
7.          $\boxed{data[num] = \text{‘A’}}$
8.          $++num$
9.          **wait** $(4, SC\_NS)$
10.     **return**

11. **process** P$_2$()
12.     **while** $(i <$ MAX$)$
13.         $\boxed{data[num] = \text{‘B’}}$
14.         $++num$
15.         **if** $(timer)$
16.             $timer = false$
17.             **notify** $(e)$
18.         **wait** $(4, SC\_NS)$
19.     **return**

20. **process** C$_1$(int $x$)
21.     **while** $(i <$ MAX$)$
22.         **if** $(num == 0)$
23.             $timer = true$
24.             $i++$
25.             **wait** $(e)$
26.         $num--$
27.         **assert** $(num \geq 0)$
28.         $\boxed{c = data[num]}$
29.         $i++$
30.         **wait** $(x, SC\_NS)$
31.     **return**
```

Figure 4.1: Simple producer-consumer example

4.5 SystemC Example

Let us start by examining the salient features of SystemC using a simple producer-consumer example shown in Figure 4.1. A SystemC program is a set of interconnected modules communicating through channels using *transactions*, events and shared variables collectively called *communication objects*. A module comprises of a set of ports, variables, processes and methods. Processes are small pieces of code that run concurrently with other processes and are managed by a *non-preemptive scheduler*. The semantics of concurrency is *cooperatively multitasking*: a type of multitasking in which the process currently executing must offer control to other processes. As such, a wait-to-wait block in a process is atomic. The processes exchange data between themselves using shared variables (signals and non-signals). During the execution of a SystemC design, all signal values are stable until all processes reach the waiting state. When all processes are waiting, signals are updated with the new values (see Update Phase in Section 4.6). In contrast, the non-signal variables are standard C++ variables which are updated immediately during execution.

For clarity the syntactic details of SystemC are not shown in Figure 4.1. It has three processes namely P_1 (lines 5-10), P_2 (lines 11-19) and C_1 (lines 20-31). The global variables of the program are shown in lines 1-4. The program uses a shared *data* array as a buffer, and an integer *num*, which indicates the total number of elements in the buffer. The producer P_1 in a loop writes to the buffer and then syncronizes by waiting (or blocking) (line 9) on time for 4 nanoseconds (*SC_NS*). Similarly, producer P_2 writes to the buffer and if *timer* is set then notifies the event e and then synchronizes using time. The consumer C_1 on the other hand, waits (or blocks) (line 25) on the event e when the buffer is empty, until the notify (line 17) on e is invoked in the P_2 process. If there are elements in the buffer then C_1 consumes it and synchronizes on time like the other processes. For synchronization SystemC uses wait-notify on events and time. In what follows, we will use this example to guide our discussion.

4.6 SystemC Simulation Kernel

Simulation involves the execution of a discrete event scheduler, which in turn triggers or resumes the execution of processes within the application. The functionality of the scheduler (as per IEEE std. [Ini05]) can be summarized as follows:

1. *Initialization Phase:* Initialize every eligible method and thread process instance in the object hierarchy to the set of runnable processes.

2. *Evaluation Phase:* From the set of runnable processes, select a process instance in an unspecified order and execute it non-preemptively. This can, in turn, notify other events, which can result in new processes being ready to run. Continue this step till there are processes to run.

3. *Update Phase:* Update signal values for all processes in step 2, that requested for it.

4. *δ-Notification Phase:* Trigger all pending δ-delayed notifications, which can wake up new processes. If, at the end of this phase, the set of runnable processes is non-empty, go back to the evaluation phase.

5. *Timed-Notification Phase (τ):* If there are pending timed notification, advance simulation time to the earliest deadline. Determine the set of runnable processes that can run at this time and go to step 2. Otherwise, end simulation.

To simulate synchronous concurrent reactions on a sequential computer SystemC supports the concept of δ-cycle. A δ-cycle is an event cycle (consisting of evaluate, update and δ-notification phase) that occurs in 0 simulation time.

Nondeterminism

For a given input, a SystemC program can produce different output behavior due to nondeterministic scheduling. To illustrate this let us consider the processes P_1 and C_1 from the example in Section 4.5 with $\mathsf{MAX} = 2$ and $x = 4$ (line 20). It has the following 4 possible executions, where τ denotes a time elapse:

- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1$ and $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a successful termination of the program with 2 A's being produced and consumed.

- $\mathsf{P}_1\mathsf{C}_1\tau\mathsf{C}_1\mathsf{P}_1$ leads to a deadlock situation. As C_1 is waiting for the event e (line 25) and P_1 has terminated.

- $\mathsf{C}_1(\mathsf{P}_1\tau)^*$ leads to an array bound violation as C_1 waits for the event e and P_1 goes on producing in the array *data*.

In general, a simulator will execute only one of the 4 possible executions. For instance with the reference OSCI simulation kernel [Ini05], only the first execution will be scheduled and the other buggy executions will be ignored. Thus, it is important to test all possible execution of a SystemC design.

Now consider the same example with all 3 processes and $\mathsf{MAX} = 8$ and $x = 2$ (line 20). It has 3701 possible executions. A naive algorithm will try to explore all possible executions one by one and will face scalability issues. In the following sections we discuss our approach of exploring these executions and how we adapt POR techniques for SystemC. For this example, our approach will explore only 767 executions and still remain *provably sufficient for detecting deadlocks and assertion violations.*

4.7 Happens-Before Relation

In this section, we describe some standard definitions used in the context of POR [HMMCM06, FG05, God95], which have been adapted here for SystemC.

We represent the behavior of a SystemC program using a state transition system $\mathcal{M} = (\Sigma, \sigma_0, \Delta)$ (see Definition 2). Recall that a transition moves a system from one state to a subsequent state and $\mathcal{T}$ denote the set of all transitions of the system. In SystemC there are three types of transitions:

1. *Immediate-transition* change the state by executing a finite sequence of operations of a chosen process followed by a *wait* operation or termination of the same process.

2. *δ-transition* change the state by updating all the signals, and by triggering all the δ-delayed notification that were requested in the current δ-cycle.

3. A *time-transition* change the system state by updating the simulation time.

In SystemC two transitions of the same process cannot be co-runnable (Definition 3).

Our goal is to explore all possible traces of the system $\mathcal{M}$. However, $\mathcal{M}$ typically contains many traces that are simply different execution order of uninteracting transitions that leads to the same final state. This observation has been exploited by partial order reduction techniques to explore a subset of the possible traces, while still being provably sufficient for detecting deadlocks and assertion violations as shown in [God95]. The following definition states the condition when two transitions are independent, meaning that they result in the same state when executed in different orders.

Definition 9 (Independence Relation). *A relation $\mathcal{I} \subseteq \mathcal{T} \times \mathcal{T}$ is an independence relation of $\mathcal{M}$ if $\mathcal{I}$ is symmetric and irreflexive and the following conditions hold for each $\sigma \in \Sigma$ and for each $(t_1, t_2) \in \mathcal{I}$:*

1. *if $t_1, t_2 \in runnable(\sigma) \quad \wedge \quad \sigma \xrightarrow{t_1} \sigma'$ then $t_2 \in runnable(\sigma')$*

2. *if $t_1, t_2 \in runnable(\sigma)$, then there is a unique state σ' such that $\sigma \overset{t_1 t_2}{\Longrightarrow} \sigma' \wedge \sigma \overset{t_2 t_1}{\Longrightarrow} \sigma'$*

Transitions $t_1, t_2 \in \mathcal{T}$ are *independent* in $\mathcal{M}$ if $(t_1, t_2) \in \mathcal{I}$. Thus, a pair of independent transitions cannot make each other runnable when executed and runnable independent transition commute. The complementary dependence relation $\mathcal{D}$ is given by $(\mathcal{T} \times \mathcal{T}) - \mathcal{I}$.

Two traces are said to be *equivalent* if they can be obtained from each other by successively permuting adjacent independent transitions. Thus, given a valid independence relation, traces can be grouped together into *equivalence classes*. For a given trace, we define a *happens-before* relation between its transitions as follows:

Definition 10 (Happens-before)**.** *Let $\phi = t_0 \cdots t_n$ be a trace in $\mathcal{M}$. A happens-before relation $\prec_\phi$ is the smallest relation on $\{0 \cdots n\}$ such that*

1. *if $i \leq j$ and $(\phi_i, \phi_j) \in \mathcal{D}$ then $i \prec_\phi j$.*

2. *$\prec_\phi$ is transitively closed.*

We use a variant of the above happens-before relation which is defined as follows: for a given trace $\phi = t_0 \cdots t_n$ in $\mathcal{M}$ and $i \in \{0 \cdots n\}$, i happens-before process P, written, $i \prec_\phi P$ if either

1. $\mathsf{Process}(\phi_i) = P$ or

2. $\exists k \in \{i+1, \cdots, n\}$ such that $i \prec_\phi k \quad \wedge \quad \mathsf{Process}(\phi_k) = P$.

4.8 Our Approach

We obtain partial-order of *runnable* processes statically by identifying the dependent transitions. A transition in SystemC is an atomic block, which in turn is a non-preemptive sequence of operations between *wait* to *wait*. Note, due to branching within an atomic block, such blocks may not be derived statically. An atomic execution is *dependent* on another atomic execution if it is enabled or disabled by the other or there exists read-write conflicts on the shared variable accesses in these blocks. In our approach, we first derive wait-notify control skeleton of the SystemC design, and then enumerate all possible atomic blocks. We then perform dependency analysis on the set of atomic blocks, and represent the information symbolically. These static information are used later, while exploring the different executions of the design. In particular, we *query* to check if a given pair of atomic blocks (corresponding to the runnable processes) need to be interleaved. If not, we do not consider that interleaving of runnable processes. In the following sections we describe our algorithm in more detail.

4.8.1 Static Analysis

Our goal is to execute only one trace from each equivalence class for a given dependence relation. Thus, the first step is to compute this dependence relation. We use static analysis techniques to compute if two transitions are dependent. Intuitively, two transitions are dependent if they operate on some shared communication objects. In particular, we use the following rules to compute the dependence relation $\mathcal{D}$, i.e. $\forall\, t_1, t_2 \in \mathcal{T}, (t_1, t_2) \in \mathcal{D}$ if any of the following holds:

1. a write on a shared *non-signal* variable v in t_1 and a read or a write on the same variable v in the other transition t_2.
2. a write on a shared *signal* variable s in t_1 and a write on the same variable s in t_2.
3. a wait on an *event* e in t_1 and an immediate notification on the same event e in t_2.

Note here that the order in which the statements occur within a transition does not matter. For each transition $t \in \mathcal{T}$, we maintain four sets - read and write sets for shared non-signal variables and shared signal variables (written, $R_{t,ns}, W_{t,ns}, R_{t,s}, W_{t,s}$ respectively). Thus, rule 1 can be re-written as, $(W_{t_1,ns} \cap R_{t_2,ns}) \cup (W_{t_1,ns} \cap W_{t_2,ns}) \neq \emptyset$. And rule 2 can be re-written as, $W_{t_1,s} \cap W_{t_2,s} \neq \emptyset$.

In the rules mentioned above, in general, two transitions with write operations on a shared variable are dependent. But to exercise more independency we consider special cases of write operations (called *symmetric write*) that can be considered as being independent (applying Definition 9). For instance, two constant addition or constant multiplication with the same variable can be considered as being independent. We also use static slicing techniques to remove irrelevant operations to further extract more independency between the transitions [HDZ00]. Intuitively, if a statement does not influence the property that we are checking than that statement can be removed in the sliced program.

To illustrate the above rules, consider the example from Figure 4.1. Consider the *wait* to *wait* atomic transition consisting of the lines (6-9) in process P_1 and the transition consisting of the lines (12-18) in process P_2. In general, these two transitions are dependent because they both write to the variable *data* and *num*. However, if the property that we are checking is the assertion in line 27 then we can get a sliced program by removing the statements inside the boxes, while still remaining correct for detecting the assertion violation. Now, if we consider only the rules 1, 2 and 3 from above then

```
1.  type $Runnable$ := list of $Transition$
2.  type $TSet$ := set of $Transition$
3.  type $State$ := $Runnable \times TSet \times TSet$
4.  type $Schedule$ := sequence of $State$

5.  function Explore() : void
6.    let $sched$ := Simulate($\emptyset$)
7.    let $depth$ := $sched$.Size $- 1$
8.    while $depth \geq 0$ do
9.       let $t$ := $sched$.Trace
10.      let $\sigma$ := $sched$.At($depth$)
11.      Analyze($t$, $depth$)
12.      if $\exists t \in \sigma$.Todo $\setminus$ $\sigma$.Sleep then
13.         $\sigma$.Runnable.Add($0, \sigma$.Runnable.Remove($t$))
14.         let $newSched$ := $sched$.Copy($0, depth$)
15.         $sched$ := Simulate($newSched$)
16.         $depth$ := $sched$.Size $- 1$
17.      else
18.         $depth$ := $depth - 1$
19.   return
```

Figure 4.2: The Explore algorithm

the two transitions are still dependent in the sliced program because they both write to the variable *num*. But, notice that both the writes to the variable *num* are symmetric (increment). Thus, we have that the two transitions are independent if the property that we are checking is only the assertion ($num \geq 0$) at line 27.

4.8.2 The Explore Algorithm

Our Explore algorithm shown in lines 5-19 of Figure 4.2, explores a reduced set of possible executions of a SystemC design. Our algorithm presented here is *stateless* [God97], i.e., it stores no state representations in memory but only information about which transitions and traces have been executed so far. Although, our approach will be slower than an algorithm that maintains full state information, it requires considerably less amount of memory, especially when the design has a large number of variables. It explores each *non-equivalent* trace of the system by re-executing the design from its initial state.

The algorithm maintains a *sched* of type *Schedule*. A *Schedule* is a sequence of *States*. Each *State* s is a 3-tuple (*Runnable*, *Todo*, *Sleep*) where, *Runnable* is a sequence of *Transitions* that are runnable in state s, *Todo* is a set of *Transitions* that needs to be explored from s, and *Sleep* is the set of *Transitions* that are no longer needed to be explored from s. The algorithm also uses a function Simulate (not shown here) that takes as input a prefix schedule and then executes it according to the trace corresponding to the schedule. Once the prefix trace ends, it randomly chooses a runnable transition that is not in the *Sleep Set* of the current state and executes it. The function continues the above step till completion of the simulation and returns the *Schedule* for the current execution. To further reduce the explored transitions, the Simulate function maintains a sleep set for each state in the same way as explained in VeriSoft [God95, God97].

The Explore function starts by executing a random schedule (as the prefix trace is $\emptyset$) and returns the schedule in *sched* (line 6). Our algorithm traverses the execution-tree bottom up and *depth* maintains the position in the tree such that the sub-tree below *depth* has been fully explored. Note that by traversing the execution-tree bottom-up we need to maintain very little state information. While we have not traversed the entire execution-tree, let $sched = s_0, \cdots s_{depth}, \cdots s_n$ then $\phi = t_0, \cdots t_i, \cdots t_{n-1}$ (line 9) is the trace corresponding to *sched* such that $\phi_i = s_i.\mathsf{Runnable.At}(0)$ and $s = s_{depth}$ (line 10). Using the computed trace ϕ, Explore then finds out the transitions that can be dependent with the transition ϕ_{depth} using the function Analyze (line 11) and adds those in the *Todo* set of the corresponding state. Next, if there exists any transition $t \in Todo \backslash Sleep$ in the state s (line 12), then the Explore function swap the transition t with the first element of *Runnable* in state s (line 13), copies the prefix schedule (line 14) and simulate it using the Simulate function (line 15). Otherwise, we have explored all required transitions in the sub tree below *depth* and now will explore all the transitions in the sub tree below $depth - 1$ (line 18). Our algorithm traverses the execution-tree bottom up and *depth* maintains the position in the tree such that the sub-tree below *depth* has been fully explored. Note that by traversing the execution-tree bottom-up we need to maintain

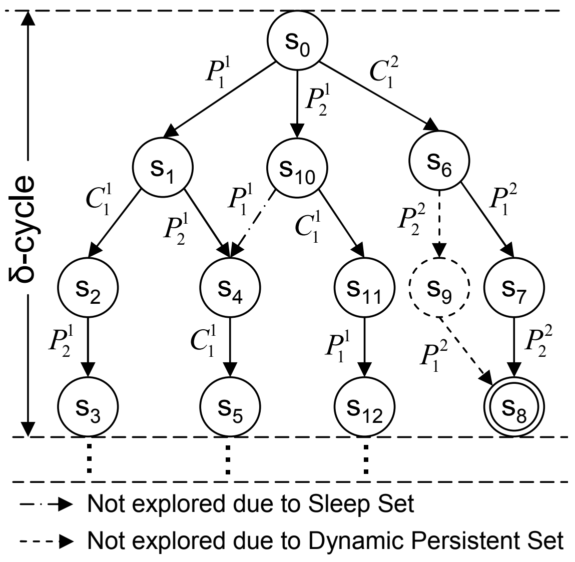
```
20. function Analyze($t$ : $Trace$, $depth$ : int) : void
21.     let $start := \mathsf{StartOfDeltaCycle}(t_{depth})$
22.     for each $i \mid start \leq i < depth$ do
23.         if $\mathsf{Query}(t_i, t_{depth}) = \mathsf{Dependent}$
24.             and $\mathsf{CoRunnable}(t_i, t_{depth}))$ then
25.             let $\sigma := \mathsf{Pre}(t, i)$
26.             let $p := \mathsf{Process}(t_{depth})$
27.             if $\mathsf{Runnable}(\sigma, p)$ then
28.                 $\sigma.\mathsf{Todo} := \sigma.\mathsf{Todo} \cup \{\mathsf{Transition}(\sigma, p)\}$
29.             elseif $\exists\, j > i \mid \mathsf{Runnable}(\sigma, \mathsf{Process}(t_j))$
30.                         and $j \prec_{t_{0 \cdots depth}} p$ then
31.                 $\sigma.\mathsf{Todo} := \sigma.\mathsf{Todo} \cup \{\mathsf{Transition}(\sigma, \mathsf{Process}(t_j))\}$
32.             else
33.                 $\sigma.\mathsf{Todo} := \sigma.\mathsf{Runnable}$
34.     return
```

Figure 4.3: The Analyze function

very little state information.

The Analyze function shown in lines 20-34 of Figure 4.3 takes as argument a trace ϕ and an integer *depth*. Next, it finds the start of the δ-cycle to which ϕ_{depth} belongs (line 21). Then, for each transition ϕ_i such that $i < depth$ and belongs to the same delta cycle (line 22), we check if ϕ_i and ϕ_{depth} are dependent using a query function (line 23) and may be co-runnable (line 24). If true, then it computes the state $s = \mathsf{Pre}(\phi, i)$ and p as the process to which the transition ϕ_{depth} belongs. Next, if there exists a transition of p that is runnable at s (line 27) then it adds that transition to the *Todo* set of s (line 28). Else, if there exists $j > i$ such that $j \prec_{t_{0 \cdots depth}} p$ (see Definition 10) and the runnable set of s contains a transition that belongs to the process to which ϕ_j belongs (line 30) then it adds that transition to the *Todo* set of s (line 31). Otherwise, it adds all runnable transitions to the *Todo* set of s (line 33).

To review our approach, consider the example from Figure 4.1 with all 3 processes and $\mathsf{MAX} = 1$ and $x = 2$ (line 20). A partial execution-tree for this example consisting of



$$\boxed{\begin{array}{c}\mathcal{I} = \{(P_2^1, P_1^1), (P_1^1, P_2^1), (P_2^2, P_1^2), (P_1^2, P_2^2)\} \\ \mathcal{D} = (\mathcal{T} \times \mathcal{T}) \backslash \mathcal{I}\end{array}}$$

Figure 4.4: A partial execution-tree showing only the first δ-cycle

only the first δ-cycle is shown in Figure 4.4. The j^{th} transition of the process $Proc$ is given by $Proc^j$. In particular, Figure 4.4 shows the following *wait* to *wait* atomic transitions P_1^1 (lines 6-9), P_1^2 (lines 6, 10), P_2^1 (lines 12-18), P_2^2 (lines 12, 19), C_1^1 (lines 21, 22, 26-30) and C_1^2 (lines 21-25). Using static analysis (as explained in Section 4.8.1), we obtain the independence relation $\mathcal{I} = \{(P_2^1, P_1^1),\ (P_1^1, P_2^1),\ (P_2^2, P_1^2), (P_1^2, P_2^2)\}$. Intuitively, if two transitions t_1 and t_2 are such that $(t_1, t_2) \notin \mathcal{I}$, then t_1 and t_2 are dependent. We use slicing of *data* and symmetric writes on *num* to determine the independency relation.

For a given data-input, let ϕ be a *trace* $(P_1^1, C_1^1, P_2^1, \cdots)$ and its corresponding state sequence be $(s_0, s_1, s_2, s_3, \cdots)$. Using the trace ϕ, we present an overview of our algorithm to explore all possible behaviors of a design for a given data-input. The state s_0 is the initial state with three runnable processes, i.e., P_1, P_2, C_1. Our Explore algorithm examines the current trace bottom up and restrict its analysis for adding backtracking points within a δ-cycle. Intuitively, for every state s_i, it checks if the transition ϕ_i, which is executed from state s_i is dependent on any other transition ϕ_j for $j < i$, i.e., in its prefix trace, that belongs to the same δ-cycle. If true, then it finds the runnable transition t_k in the pre-state s_j of ϕ_j (see trace in Definition 4), which has a causal order with ϕ_i

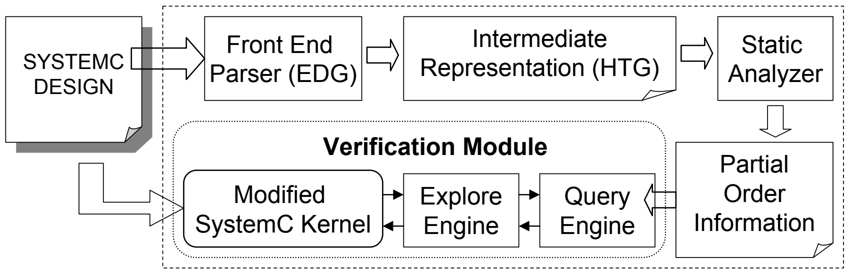


Figure 4.5: Our prototype Satya tool

and adds t_k to the backtracking set of s_j. For example, when the algorithm examines the state s_2, it adds P_2^1 to the backtracking set of s_1 (since, P_2^1 and C_1^1 are dependent). Next, when it analyzes the state s_1 the algorithm adds C_1^2 to the backtracking set of s_0 and then explores the trace $\psi = (P_1^1, P_2^1, C_1^1, \cdots)$ (as P_2^1 was in s_1's backtracking set). Next, it analyzes the new trace ψ in a similar fashion. The algorithm continues in this way, till it reach state s_7, at this point P_2^1 is added to the backtracking set of s_0. The transition and state shown in dashed line is not explored. The state s_8 is a deadlock state. Note that the transition P_1^1 is not explored in the state s_{10} because it is in the *Sleep* set of s_{10} (as P_1^1 and P_2^1 are independent). Our algorithm explores only 4 different traces out of the 8 possible traces for this example.

4.9 The Satya Tool

We implemented our algorithm to explore all possible valid traces of a SystemC design in a prototype tool called Satya. The implementation of Satya consists of 2 main modules - a static analyzer and a verification module. The Satya software tool is over 18,000 lines of C++ code and uses the EDG C++ front-end parser [(ED92] and the OSCI SystemC simulator [Ini05]. Of those, about 17,500 lines are the intermediate representation (IR) and utility functions needed by the static analyzer and the verification module (explore and query engine) is only about 800 lines.

Figure 4.5 presents an overview of the Satya tool. It takes a SystemC design as input - currently with the restriction of no dynamic casting and no dynamic process creation. After parsing the design description, we capture the EDG intermediate language into our own IR that consists of basic blocks encapsulated in Hierarchical Task Graphs (HTGs) [GP92]. We choose HTG over other IR's like CDFG, as HTG maintains the heirarchical structuring of the design such as if-then-else blocks and for- and while-loops which are used during static analysis. The static analyzer work on the HTGs in a compositional manner to generate the dependency relation, which is then used by the query engine. The implementation of the explore engine follows closely the algorithm described in Section 4.8.2.

The SystemC design is compiled with the verification module which contains a modified OSCI's SystemC kernel. The modified kernel implements the Simulate function of our Explore algorithm (Figure 4.2). It takes as input a prefix schedule and executes it till completion such that the prefix of the executed trace is same as the trace corresponding to the input prefix schedule. The modifications are still in compliance with the SystemC specification [Ini05]. In particular, the modifications are to replace the election algorithm of the scheduler by one, which takes as input a prefix trace that acts as a golden reference for that run and execute it till completion.

4.10 Experiments and Results

Using Satya, we experimented on several small examples and two benchmark designs. In this section, we discuss the results for the two benchmarks.

4.10.1 FIFO Benchmark

The first benchmark is a FIFO channel example obtained from the OSCI's example repository [Ini05]. The example has an hierarchical channel FIFO. To use the FIFO channel it uses a producer-consumer scenario. The example works fine when executed in one producer and one consumer scenario. However, if we use two producers writing to the channel and one consumer reading from that channel then we have an assertion violation. Moreover, since this bug is not visible in every trace of the example, simulation may not find it. Our tool was able to find the bug and consequently we changed the code to correct it. The following results are measured on the corrected example. The example has 3 processes executing concurrently. The total number of possible traces is directly proportional to the number of elements produced by the producers. To quantify the scalability of our tool, we report in Table 4.1 for different number of elements produced by the two producers, the time required using POR and the number of reduced traces explored by our tool, along with the total number of possible traces and the time required without POR.

Table 4.1: Results for the FIFO benchmark

Elements produced	Reduced #traces	Time (POR) sec:msec	Total #traces	Time (no-POR) sec:msec
14	6	00:032	8	00:046
28	42	00:265	80	00:469
44	318	02:313	992	06:344
62	2514	19:031	13376	93:563

4.10.2 TAC Benchmark

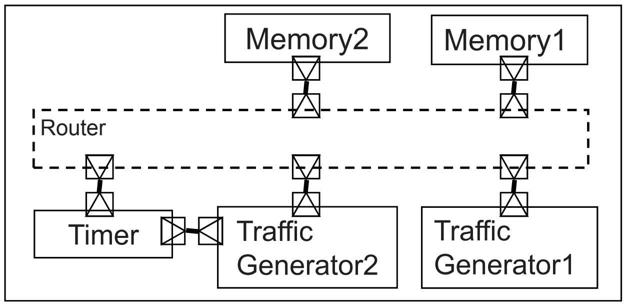


Figure 4.6: SystemC TAC benchmark

The second benchmark is the industrial Transaction Accurate Communication (TAC) example [Mic05] developed by ST Microelectronics, which includes a platform composed of the following 6 modules: two traffic generators, two memories, a timer and a router to connect them as shown in Figure 4.6. These modules are based on the TAC protocol built on top of OSCI's TLM standard. This benchmark is over 12,000 lines of SystemC code and consists of 349 functions. The example can be executed for certain number of transactions. A transaction is a read or write by the masters, namely the two traffic generators. When executed for 80,000 transactions, there are 12032 total possible traces. It took 89.47 minutes to explore all these traces, whereas while checking for deadlocks in the program, we did a static slicing of the router and then our tool found all these traces to be equivalent to only one trace, which was executed in 1.3 seconds.

It is interesting that all the different traces of the program is equivalent to one trace for this case. This is because the way this benchmark is written the traffic generator1 writes only to memory1 and traffic generator2 writes only to memory2, and as such there are no conflict between them. Furthermore, for this example simulation has same coverage as our tool, however simulation cannot provide correctness guarantee that is provided by our tool.

4.11 Related Work

SystemC Verification: Prior work on SystemC focuses mainly on improving simulation performance [MVD03] and generating representative inputs for the design [GED07] and formalizing the semantics of SystemC [MMMC05, KS05, HT05, SDGK07], while ignoring the problem of generating all possible behaviors of the design. However, recently researchers address the above problem by automatically generating all valid scheduling of the design [HMMCM06]. Their work used dynamic partial-order reduction (POR) techniques [FG05] to avoid generation of two schedulings that have the same effect on the system's behavior.

Partial-Order Reduction: POR techniques are extensively used by *software model checkers* for reducing the size of the state space of concurrent system at the implementation level [God97, Hol97]. Other state space reduction techniques, such as slicing [HDZ00, SG07] and abstraction [BMMR01], are orthogonal and can be used in conjunction with POR. The POR techniques can be divided in two main categories: *static* [God95] and *dynamic* [FG05].

The main static POR techniques are *persistent/stubborn* sets and *sleep* sets [God95]. Intuitively, the persistent/stubborn set techniques compute a provably sufficient subset of the enabled transitions in each visited states such that if a selective search is done using only the transitions from these subsets the detection of all the deadlocks and safety property violations is guaranteed. All these algorithms infer the persistent sets from the static structure (code) of the system being verified. On the other hand, the sleep set techniques exploits independences between the transitions in the persistent sets to reduce interleavings. Both these techniques are orthogonal and can be applied simultaneously [God95]. In contrast, the dynamic POR technique evaluates the dependency relation dynamically between the enabled and executed transitions for a given execution.

4.12 Summary

In this chapter, we have presented a scalable approach for testing SystemC designs. We have implemented it in a query-based tool called Satya. Our approach combines static and dynamic POR techniques to reduce the number of interleavings required to expose all behaviors of SystemC. Furthermore, our approach exploits SystemC specific semantics to reduce the number of backtracking points, and thereby improving the efficiency of the approach. Our experiments on a set of examples show the efficacy of our approach and finds bugs that may not have been found using a simulator. In the next chapter we will discuss another property checking approach which is based on symbolic model checking.

Acknowledgements: Chapter 4, in part, has been published as "Partial Order Reduction for Scalable Testing of SystemC TLM Designs" by Sudipta Kundu, Malay Ganai and Rajesh Gupta in *DAC 08: Proceedings of the 45th annual conference on Design Automation* [KGG08]. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Efficient Symbolic Analysis for Concurrent Programs

In this chapter, we describe a method for verifying concurrent programs based on symbolic approach as a part of our high-level property verification strategy. In the previous chapter, we discussed a technique that explores the state space of a program by explicit enumeration. In contrast, the technique in this chapter manipulate sets of states, instead of individual states. This algorithm avoids explicitly building the complete state graph for the system; instead, they represent the state graph implicitly using a formula in propositional logic. Furthermore, in this chapter we will discuss some more reduction techniques including partial-order reduction for symbolic algorithms.

5.1 Synchronous vs. Asynchronous

Based on how verifications models are built, symbolic approaches can be broadly classified into: *synchronous* and *asynchronous* modeling. Both these methods have inherent advantages and disadvantages, in regards, to applicability of state-reduction and state-representation for addressing scalability.

Synchronous modeling: In this category of symbolic approaches [ABH$^+$01, KGS06], a synchronous model of concurrent programs is constructed with a scheduler. The scheduler is then constrained – by adding more constraints, i.e., guard strengthening – to explore only a subset of interleaving. To guarantee correctness, the scheduler must allow context-switch between accesses that are conflicting. For scalability issues, one determines statically (i.e., conservatively) which pair-wise locations require context switches, using persistent [God95] set computations. One can further use lock-set and/or lock-acquisition history analysis [SC06, FQ03, KGS06], and conditional dependency [GP93, WYKG08] to reduce the set of interleavings need to be explored.

Asynchronous Modeling: In this category, the symbolic approaches such as TCBMC [RG05] and token-based [GG08] generate verification conditions directly without constructing a synchronous model of concurrent programs, i.e., without using a scheduler. The concurrency constraints that maintain interleaving semantics are included in the verification conditions on-the-fly for a bounded depth analysis.

Relative Strengths: In *synchronous* modeling approaches, underlying model checker used is a standard one, typically geared for synchronous hardware models. Most reduction techniques for these approaches add more constraints to reduce the search space of the program. *Asynchronous* modeling approaches, on the other hand, uses customized model checkers built for software programs. In these techniques, first, the constraints are added to allow necessary interleavings, and then reduction techniques remove some of the redundant constraints. It turns out, for systems with a small interleavings, the latter suits better than the former as it reduces size of the verification conditions.

5.2 Overview of Our Approach

We present a new symbolic method combining partial order reduction with an asynchronous modeling approach to reduce verification problem size and state space for multi-threaded concurrent system with shared variables and locks. To our knowledge so far, the partial-order reduction has *hardly* been exploited in the asynchronous modeling approaches [RG05, GG08].

We combine our method with a previous token-based approach [GG08] that otherwise allows redundant interleavings. We introduce the notion of *Mutually Atomic Transactions* (MAT), i.e., two transactions are mutually atomic when there exists exactly one conflicting shared-access pair between them. We propose to reduce the verification conditions *and* remove redundant interleavings by allowing pair-wise (token-passing) constraints *only* between MATs, i.e., token can pass from the end of one transaction to the beginning of the other (and not in between), and vice versa.

We exploit static analysis techniques based on fork-join structure and lock set information to further reduce the necessary pair-wise constraints. Furthermore, we seamlessly integrate context-bounding [QR05] in our approach. Context bounding is an incomplete technique that bounds the number of context switches allowed by an interleaving. Context switch is defined in Operating Systems as the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource. Researchers have shown that most concurrency bugs can be found within a small number of context switches [LR08, Mus08, QR05]. Context bounding is an effective technique for bug finding, while still remaining scalable.

We implemented our approach in a prototype tool called Candor, and demonstrated the efficacy of our approach against the state-of-the-art symbolic approaches [GG08, WYKG08]. In this chapter, we will discuss the implementation of the various parts of our algorithm.

Contributions

Our main contributions can be summarized as follows:

1. We are first to exploit partial order reduction techniques for symbolic model checking using asynchronous modeling. We developed a novel approach – based on MAT – to reduce verification problem size and state space for concurrent systems.

2. Our approach based on MAT, exploits simultaneous unreachability of conflict access – due to happens-before relation such as fork-join and mutual exclusion due to locking pattern in program structure – to further reduce pair-wise constraints, thereby integrating them easily and uniformly under one framework.

3. We efficiently encode thread-specific context bounding, and fork-join semantics in verification conditions.

4. We demonstrate that our approach outperforms previous approaches [GG08, WYKG08] in most cases, both in performance and size of the verification problem.

5. When our approach finds an error, we present the counterexample using a trace in the control-flow graph. This visual representation makes debugging of bugs easier and faster.

5.3 Illustrative Example

This section presents an overview of our approach. We start out by describing a simple concurrent program that we will use to guide our discussion. The concurrent control flow graph (CCFG) of our two-threaded example is shown in Figure 5.1. The program consists of two *unrolled* threads M_1 and M_2. The variables a_1, a_2 and b_1, b_2 of threads M_1 and M_2 respectively are local; and the variables x, y, and z are shared. The shared variables of the program are initialized with non deterministic values (represented using question mark, `?` in Figure 5.1) such that $(x > -7)$ and $(y < z + 5)$. The thread M_1 forks the thread M_2 and then joins it later on, before asserting that $y > 0$ holds. The statements associated with an edge in the CCFG is *atomic*, i.e., it cannot be interrupted. Each edge in the CCFG is further labeled with $W(v)/R(v)$ based on the shared write/read access of variable v for the statements in that edge.

Our goal is to model check the concurrent program for properties like data races, deadlocks, and assertion violations. For our example, we want to check if the assertion $y > 0$ (in Figure 5.1) is ever violated. In general, an asynchronous concurrent system can have many different possible behaviors or executions due to nondeterministic scheduling of the statements in different threads. In the following subsection we discuss an approach to efficiently model an asynchronous multi-threaded system.

5.3.1 Token-passing Model

The main idea of the token-passing approach [GG08] is to introduce a single boolean token `tk` and a clock vector variable `cs` in the system, and then manipulate the passing of the token to allow different interleavings of the concurrent system. For each thread, the clock vector variable `cs` records the number of times the token `tk` is passed by the thread. In this approach, the verification model is obtained in two phases.

In the first phase, each thread is decoupled from the other threads by localizing all the shared variables, and is abstracted by letting these localized shared variables take non-deterministic values at every shared access. The token-passing model for our example is shown in Figure 5.2. The two threads M_1 and M_2 are first decoupled by renaming each shared variable in the threads. For instance, the shared variable x is renamed to x_1 and x_2 in the threads M_1 and M_2 respectively. For clarity of presentation, unless there is an ambiguity we avoid renaming of the variables (for example, the renaming of variables is not shown in Figure 5.2). The algorithm then introduces two new control

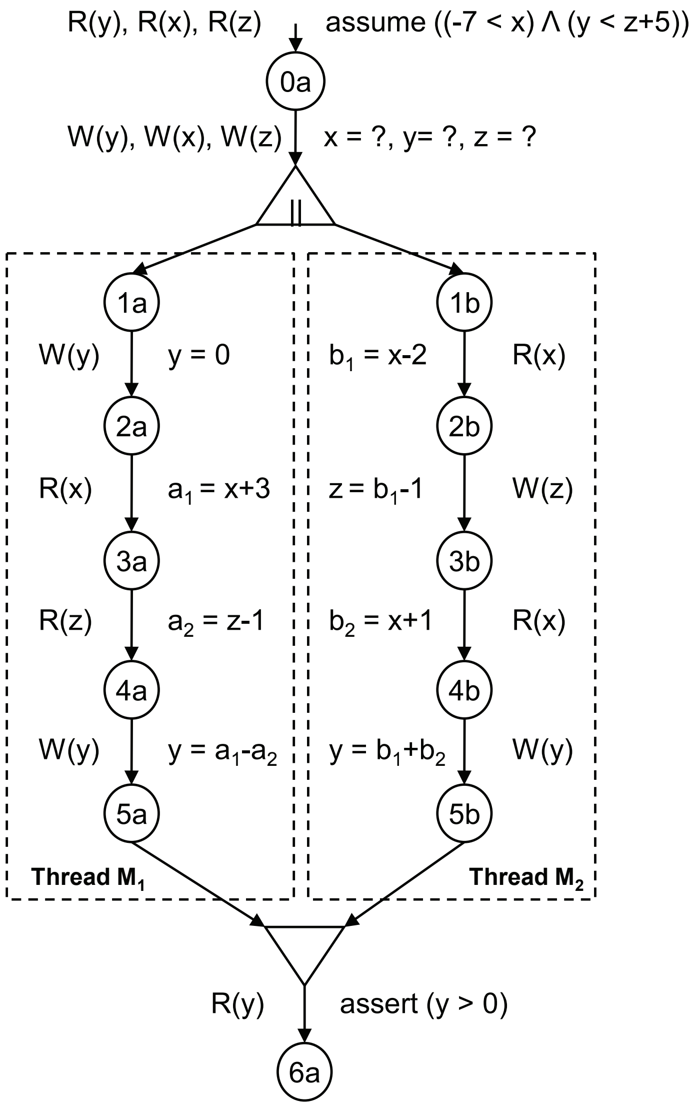


Figure 5.1: The CCFG of our running example

states (locations) for each edge (transition) with shared accesses. For each such edge, a *pre-access control state* (`pre`) is inserted before the edge and a *post-access control state* (`post`) is inserted after the edge. The new edge corresponding to the `pre` node is then initialized such that all the localized shared variables get a non-deterministic value. In our example, for clarity, we highlight such instrumentations only for the edge $(2b, 3b)$. The statements x`=?,` y`=?,` z`=?,` tk`=?,` cs`=?` are added in the edge corresponding to

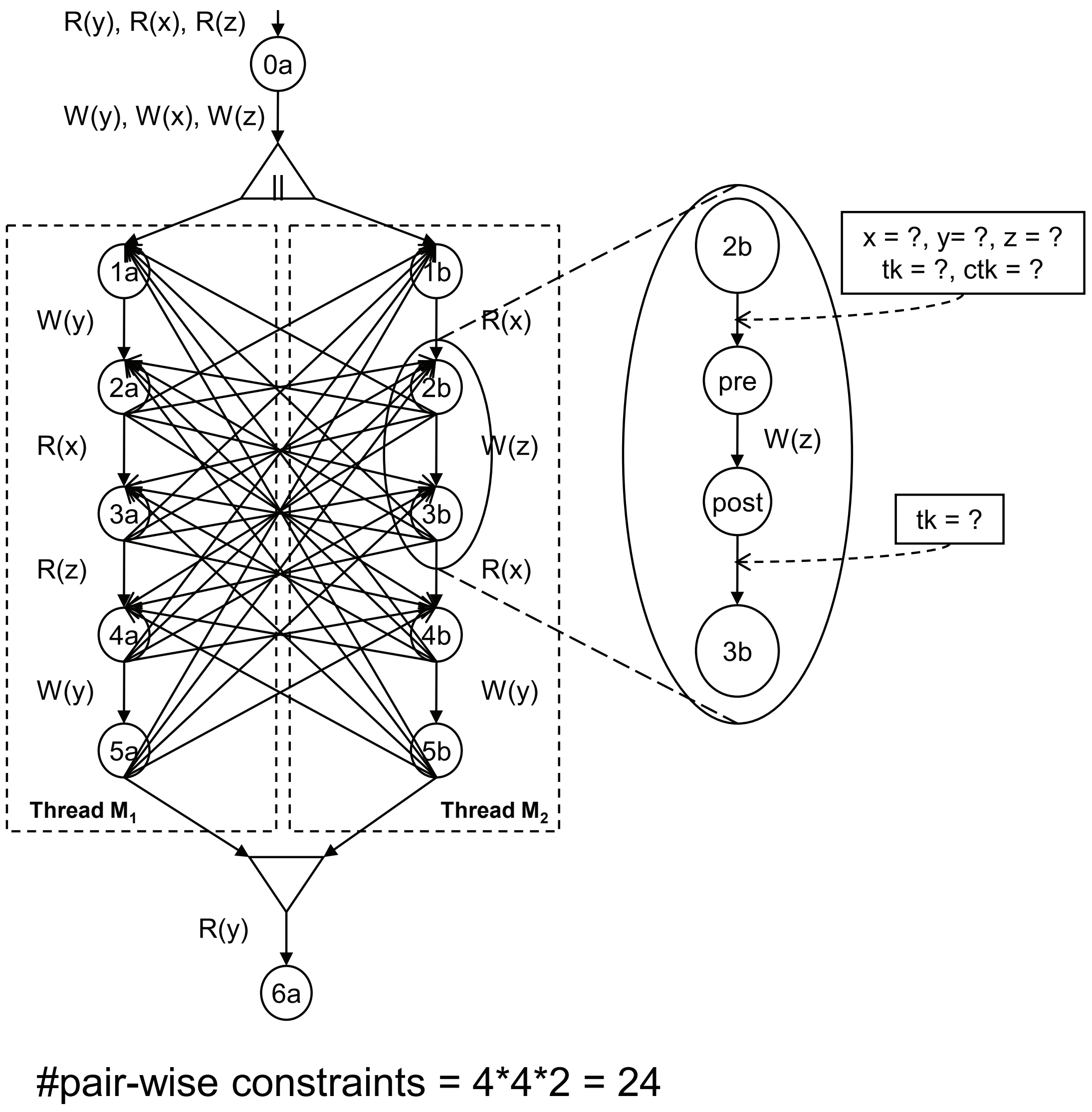


Figure 5.2: The token passing model for our running example

the `pre` node and the statement *tk*=`?` is added in the edge corresponding to the `post` node. (Recall that `?` denotes non-deterministic values.) Since the transition (update) relation only uses local variables (as shared variables have been localized), each thread model is independent. However, due to abstraction (by using non-deterministic values for the localized shared variables), the model now have additional behaviors, hence, it is imprecise.

The goal of the second phase is to remove the imprecision caused due to abstraction and also to cover all possible behavior of the program. In this phase, constraints are added on-the-fly to restrict the introduced non-determinism and to obtain a total order in the shared accesses. More specifically, for each pair of shared accesses in different threads, *token-passing constraints* are added from the `post` node of a shared access to the `pre` node of the other shared access. Intuitively, these token-passing constraints allow passing of the token from one thread to another. Furthermore, these constraints also allow to synchronize the values of the localized shared variables from one thread to another. Together the token-passing constraints restrict the choice for the non-deterministic values in order to satisfy sequential consistency (see Definition in Section 5.4).

In Figure 5.2, we denote a token-passing constraint using a directed edge from a `post` node of one thread to a `pre` node of the other. As shown in the figure such token-passing constraints are added for all pairs of shared accesses. For example, a constraint from M_1 to M_2 will have the statements x_2=x_1, y_2=y_1, z_2=z_1, tk_2=tk_1, and cs_2=cs_1 (Recall that for a shared variable v, v_i is the localized variable in the thread M_i). The above statements allow the two threads to synchronize. In this example we exploit happens-before relation due to fork-join such that token-passing constraints are not added between thread M_2 and the shared accesses before the fork or after the join. The token-passing constraints allow the model to capture all possible interleavings. Note that an execution of the concurrent system, includes the firing of only a subset of these constraints. As shown, the algorithm adds $4 * 4 * 2 = 32$ such token-passing constraints for our example. The following theorem summarizes the completeness and soundness of the token passing approach.

Theorem 1 (Ganai et al., 2008 [GG08]). *The token-based model is both complete (i.e., it allows only sequentially consistent traces) and sound (i.e., it does not miss any necessary interleaving) for a bounded depth analysis. Further, the number of token-passing constraints added grow quadratically (in the worse case) with the analysis depth.*

In the next subsection, we build our partial-order reduction approach over such a token-passing model. In particular, we identify token-passing constraints that can be safely removed (24 for our example), without affecting soundness and completeness, and maintaining optimality by not allowing redundant interleavings. Previous approaches on partial-order reduction for explicit model [Hol97, God97, Dil96, FG05, GFYS07] would not be applicable here as those techniques cannot deal with symbolic representation of the concurrent program. Also, previous partial-order reduction approaches for synchronous

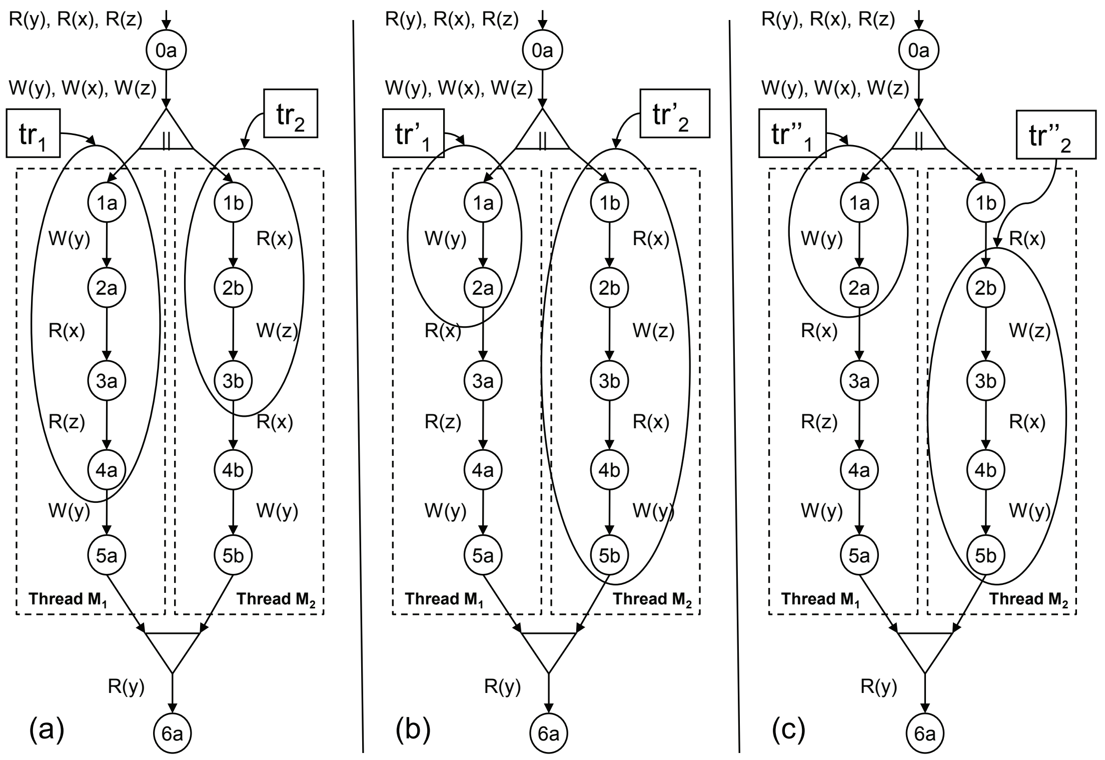


Figure 5.3: (a) $m{=}(tr_1, tr_2)$, (b) $m'{=}(tr'_1, tr'_2)$, (c) $m''{=}(tr''_1, tr''_2)$

symbolic model checking [ABH+01, KGS06, RG05, WYKG08, GG08] would not work for the token-passing approach. This is because those techniques work by constraining the scheduler to explore only a subset of interleavings, however such a scheduler is not present in the asynchronous approaches. Furthermore, in Section 5.10 we show that our approach i.e. token passing model with MAT, outperforms in most cases a state-of-the-art synchronous symbolic technique [WYKG08].

5.3.2 Mutually Atomic Transactions

Our partial-order reduction approach is based on the concept of MAT. Intuitively, let a transaction be a sequence of statements in a thread, then we say two transactions tr_i and tr_j $(i \neq j)$ of threads M_i and M_j respectively, are *mutually atomic transactions* if and only if there exist exactly one conflicting shared-access pair between them and the statements containing the shared-access pair is the last one in each of the transactions. A more formal definition of MAT is presented in Section 5.4.

Figure 5.3 illustrates the concept of MAT using our example from Figure 5.1. From the control state pair $(1a, 1b)$, there are two reachable conflicting pairs, i.e., $(4a, 3b)$ (for read-write conflict on the variable z) and $(2a, 5b)$ (for write-write conflict on the variable y), and two MATs $m = (tr_1 = 1a \cdots 4a, tr_2 = 1b \cdots 3b)$ and $m' = (tr'_1 = 1a \cdots 2a, tr'_2 = 1b \cdots 5b)$, respectively. Similarly, from $(1a, 2b)$ we have another MAT $m'' = (tr''_1 = 1a \cdots 2a, tr''_2 = 2b \cdots 5b)$. In general, although we don't show it here, there are multiple possible MATs for our example.

Given a MAT $m = (tr_i, tr_j)$, an interesting fact is that there are only *two* different program behavior possible by interleaving the various statements in the transactions. For example, consider the MAT $m = (tr_1 = 1a \cdots 4a, tr_2 = 1b \cdots 3b)$ from Figure 5.3(a), the order of execution of $(1a, 2a)$ and $(1b, 2b)$ or $(3a, 4a)$ and $(1b, 2b)$ does not produce different behaviors. In fact only the order of execution of $(3a, 4a)$ and $(2b, 3b)$ (the conflicting pair) can possibly produce different behaviors. In general, the execution of tr_i before tr_j and tr_j before tr_i succinctly captures the two possible different behaviors. Note that reordering of statements within a transaction are not allowed due to program order.

The above observation gives an intuition that there exists a set of MATs such that by adding token-passing constraints only between the MATs, we will be able to capture all possible interleavings of the system. In Section 5.7 we describe an algorithm $GenMAT$ to compute such a set of MATs. For our example one such set is $\{$ $(1a \cdots 4a, 1b \cdots 3b)$, $(4a \cdots 5a, 1b \cdots 5b)$, $(1a \cdots 2a, 3b \cdots 5b)$, $(4a \cdots 5a, 3b \cdots 5b)$, $(2a \cdots 5a, 3b \cdots 5b)\}$. In Section 5.7 we describe in details how we compute this set. Figure 5.9 shows that the total number of token-passing constraints we add are 8, much less compared with all pair-wise constraints in Figure 5.2.

5.4 Preliminaries

Having illustrated our approach using a simple example, we now present a formal description. We first describe some related definitions that we will use in the following sections. We consider a multi-threaded system $\mathcal{CS}$ comprising a finite number of deterministic bounded-stack threads communicating with shared variables, some of which are used as synchronization objects such as locks. We represent each thread model $M_i (1 \leq i \leq N)$ using a transition diagram $\pi_i = (\mathcal{L}_i, \mathcal{I}_i, \rightarrow_i, \iota_i)$ (Definition 5). Let $\mathcal{T} = \bigcup_i \rightarrow_i$ be the set of all transitions. Let V_i be set of local variables in π_i and $\mathcal{V}$

be set of (global) shared variables. A global transition diagram for $\mathcal{CS}$ is an interleaved composition of the individual thread models. Each transition consists of global firing of a local transition $t_i \in \mathcal{T}$.

Notation: We define the notion of a run of a multi-threaded program as observation of events such as global accesses, thread creations and thread termination. If the events are ordered, we call it a total order run. We define a set A_i of shared accesses corresponding to a read $R_i(x)$ and a write $W_i(x)$ of a thread M_i where $x \in \mathcal{V}$. For $a_i \in A_i$, we use $var(a_i)$ to denote the accessed shared variable. We use $\vdash_i$ to denote the beginning and $\dashv_i$ to denote the termination of thread M_i, respectively. The alphabets of events of thread M_i is a set $\Lambda_i = A_i \cup \{\vdash_i, \dashv_i\}$. We use $\Lambda = \cup_i \Lambda_i$ to denote a set of all events. A word ω defined over the alphabet set Λ, i.e., $\omega \in \Lambda^*$ is a string of alphabet from Λ, with $\omega[i]$ denoting the i^{th} access in ω, and $\omega[i,j]$ denoting the access substring from i^{th} to j^{th} position, i.e., $\omega[i] \cdots \omega[j]$ ($\cdot$ denotes concatenation). $|\omega|$ denotes the length of the word ω. We use $\pi(\omega)$ to denote a permutation of alphabets in the word ω. We use $\omega \mid_i$ to denote the projection of ω on thread M_i, i.e., inclusion of the actions of M_i only.

Definition 11 (Transaction). *A transaction is a word $tr_i \in \Lambda_i^*$ that may be atomic (i.e., uninterrupted by other thread) with respect to some other transactions. If it is atomic with respect to all other thread transactions, we refer it as* independent transaction.

Definition 12 (Schedule). *Informally, we define a schedule as a total order run of a multi-threaded program where the accesses of the threads are interleaved. Formally, a schedule is a word $\omega \in \Lambda^*$ such that $\omega \mid_i$ is a prefix of the word $\vdash_i \cdot A_i^* \cdot \dashv_i$.*

Definition 13 (Happens-before Relation ($\prec, \preceq$)). *Given a schedule ω, we say e happens-before e', denoted as $e \prec_\omega e'$ if $i < j$ where $\omega[i] = e$ and $\omega[j] = e'$. We drop the subscript if it is obvious from the context. Also, if the relation is not strict, we use the notation $\preceq$. If $e, e' \in \Lambda_i$ and e precedes e' in ω, we say that they are in a thread program order, denoted as $e \prec_{po} e'$.*

Definition 14 (Sequentially Consistent). *A schedule ω is sequentially consistent [Lam79] iff (a) $\omega \mid_i$ is in thread program order, (b) each shared read access gets the last data written at the same address location in the total order, and (c) synchronization semantics is maintained, i.e., the same locks are not acquired in the run without a corresponding release in between.*

For our purpose, we restrict our analysis to schedules that are sequentially consistent.

Definition 15 (Conflicting Access $\mathcal{C}_{ij}$)**.** *We define a pair* $a_i \in A_i, a_j \in A_j$, $i \neq j$ *conflicting, if they are accesses on the same shared variable (i.e.,* $var(a_i) = var(a_j)$*) and one of them is write access. We use* $\mathcal{C}_{ij}$ *to denote the set of tuples* (a_i, a_j) *of such conflicting accesses.*

We use Sh_{ij} to denote a set of shared variables – between M_i and M_j threads – with at least one conflicting access, i.e., $Sh_{ij} = \{var(a_i)|(a_i, a_j) \in \mathcal{C}_{ij}\}$. We define $Sh_i = \bigcup_{i \neq j} Sh_{ij}$, i.e., a set of variables shared between M_i and M_k, $k \neq i$ with at least one conflicting access. In general, $Sh_{ij} \subseteq (Sh_i \cap Sh_j)$.

Definition 16 (Dependency Relation (D))**.** *A relation* $D \subseteq \Lambda \times \Lambda$ *is a dependency relation iff for all* $(e, e') \in D$*, one of the following holds: (1)* $e, e' \in \Lambda_i$ *and* $e \prec_{po} e'$*, (2)* $(e, e') \in \mathcal{C}_{ij}$*, (3)* $e = \dashv_i$, $e' = \dashv_j$ *for* $i \neq j$*.*

Note, the last condition is required when the order of thread termination is important. If $(e, e') \notin D$, we say the events e, e' are *independent*. The dependency relation in general, is hard to obtain; however, one can obtain such relation conservatively using static analysis [God95], which may result in a larger dependency set than required. For our reduction analysis, we assume such a relation is provided.

Definition 17 (Equivalence Relation ($\simeq$))**.** *We say two schedules* $\omega_1 = u \cdot e \cdot e' \cdot v$ *and* $\omega_2 = u \cdot e' \cdot e \cdot v$ *are equivalent (Mazurkiewicz's trace theory [Maz87]), denoted as* $\omega_1 \simeq \omega_2$*, if* $(e, e') \notin D$*.*

An equivalent class of schedules can be obtained by iteratively swapping the consecutive independent events in a given schedule. Note, all equivalent schedules agrees on $e \prec e'$ if $(e, e') \in D$. Final values of both local and shared variables remains unchanged when two equivalent schedules are executed.

Definition 18 (Mutually Atomic Transactions)**.** *We say two transactions* tr_i *and* tr_j *of threads* M_i *and* M_j*, respectively, are* mutually atomic *iff except for the last pair, all other event pairs in the corresponding transactions are independent. Formally, a Mutually Atomic Transactions (MAT) is a pair of transactions, i.e.,* $(tr_i, tr_j), i \neq j$ *iff*

$$\forall k, h \;\; 1 \leq k < |tr_i| \;\; \wedge \;\; 1 \leq h < |tr_j| \;\; \wedge \;\; (tr_i[k], tr_j[h]) \notin D \;\; \wedge \;\; (tr_i[|tr_i|], tr_j[|tr_j|]) \in D$$

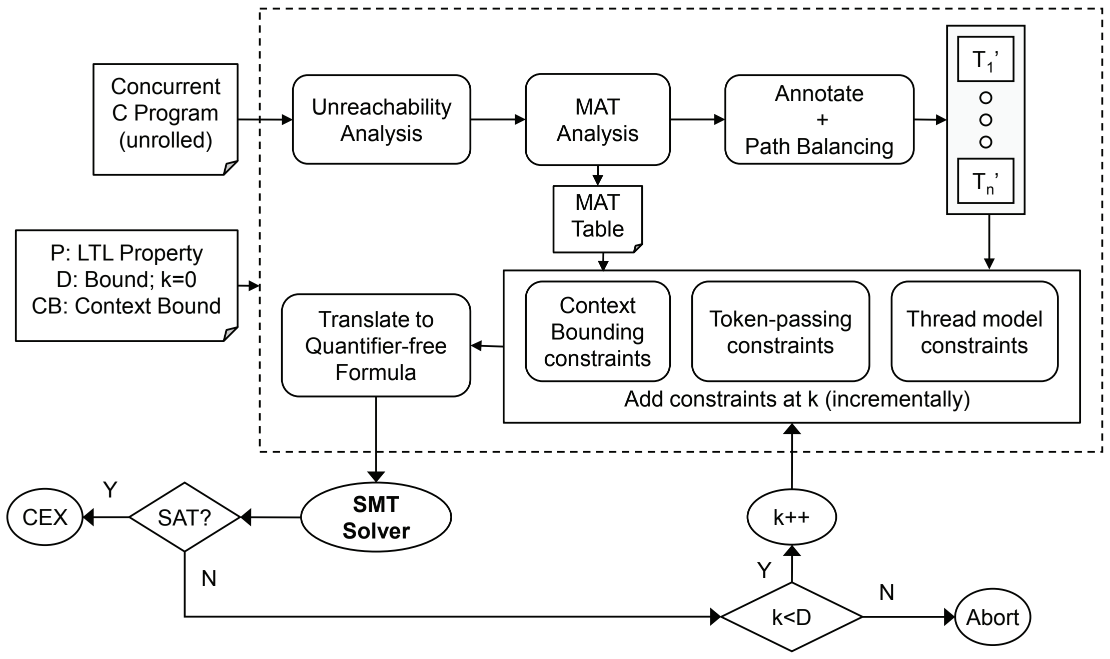

Concurrent C Program (unrolled)
P: LTL Property
D: Bound; k=0
CB: Context Bound
Unreachability Analysis
MAT Analysis
Annotate + Path Balancing
T_1'
T_n'
MAT Table
Translate to Quantifier-free Formula
Context Bounding constraints
Token-passing constraints
Thread model constraints
Add constraints at k (incrementally)
SMT Solver
SAT?
Y
CEX
N
k++
Y
k<D
N
Abort

Figure 5.4: Our SMT-based symbolic analysis tool (Candor)

As mentioned briefly in Section 5.3.2, given a MAT (tr_i, tr_j), an interesting observation is that a word $\omega = tr_i \cdot tr_j$ is equivalent to any word $\pi(\omega)$ obtained by swapping any consecutive events $tr_i[k]$ and $tr_j[h]$ such that $k \neq |tr_i|$ and $h \neq |tr_j|$. Similarly, the word $\omega' = tr_j \cdot tr_i$ is equivalent to any word $\pi(\omega')$ obtained as above. Note, $\omega \not\simeq \omega'$. Therefore, for a given MAT, there are only two equivalent classes, represented by ω and ω'. In other words, given a MAT, the associated transactions are *atomic pair-wise.*

5.5 The Candor Tool

In the following sections, we describe in details our SMT-based symbolic analysis tool called Candor. Candor is based on the token-passing model [GG08], and combines various static analysis techniques to generate a reduced verification condition. Figure 5.4 presents an overview of Candor. In the next sections we discuss the various components of Candor. Candor takes as input a unrolled concurrent program and converts it into a CCFG. Our symbolic analysis like any bounded model checking approaches is a bounded depth analysis. In Candor the unrolling of the loops in the program till the bound D is done a priori (eager), however an approach where the unrolling is done on demand basis (lazy) can also be used with our approach.

5.6 Unreachability Analysis

A concurrent program consists of a set of unrolled interacting threads $M_1 \cdots M_N$, communicating using shared variables and synchronization primitives like lock and unlock. For each pair of thread M_i and M_j, we first compute the set $\mathcal{C}_{ij}$ of pair of transitions that are in conflict (Definition 15). The conflict set $\mathcal{C}_{ij}$ in general, is hard to obtain; however, one can obtain such a set conservatively using static analysis, which may result in a larger set than required. In this component, we update the set $\mathcal{C}_{ij}$ by removing the pair that are unreachable simultaneously. In particular, we use happens-before relation due to fork-join and mutual exclusion due to lock-unlock pattern.

5.6.1 Fork-Join Analysis

We use static analysis to compute the happens-before relation due to program order and also due to fork-join. For this purpose, we annotate each transition with a pair of integers (ρ, η) where, ρ names the different path in the transition diagram and η represents the ordering within a path. The algorithm in Figure 5.5 computes the happens-before ordering of the transition diagram. The ComputeHB function performs a depth first search of the transition diagram and assigns the ρ and the η of each transition. In particular, within a path in the transition diagram, it assigns the η's with increasing values and also every path gets a different id. Since within a path the id's are increasing, we only have to store the happens-before relation between the start and the end of the paths. In ComputeHB function, we store the above information in the set *setHB* as shown in line 12 for start of path and in line 4 for end of path. The ComputeHB algorithm visits each transition exactly once and as such this analysis is bounded by the number of transitions.

We use the function $\mathsf{IsHB}(t_1, t_2)$ to check if the transition t_1 happens-before transition t_2. We then update the set $\mathcal{C}_{ij}$ as follows:

$$\mathcal{C}_{ij} = \mathcal{C}_{ij} \backslash \{(t, t') \in \mathcal{C}_{ij} \mid \mathsf{IsHB}(t, t') \ \vee \ \mathsf{IsHB}(t', t)\}$$

Example: Consider the example from Figure 5.1. The ComputeHB algorithm will assign the transitions between nodes $0a - 6a$ in thread M_1, the $\rho = 0$ and the η from

```
1.  $setHB$ : (int × int) × (int × int)
2.  **function** ComputeHB($t \in \mathcal{T}$, $\rho$ : int, $\eta$ : int) : void
3.      **if** $t.visited$ **then**
4.          $setHB$.Add($(\rho, \eta), t.(\rho, \eta)$)
5.      **let** $t.visited := true$
6.      **let** $\eta := \eta + 1$
7.      **let** $t.(\rho, \eta) := (\rho, \eta)$
8.      **let** $(\_, \_, gl') = t$  /* '_' represent variables not used */
9.      **let** $flag := false$
10.     **for each** $\{t' \in \mathcal{T} \mid t' = (gl', \_, \_)\}$ **do**
11.         **if** $flag$ **then**
12.             $setHB$.Add($t.(\rho, \eta), (\rho, \eta + 1)$)
13.         ComputeHB($t', \rho, \eta$)
14.         **let** $\rho := \rho + 1$
15.         **let** $flag := true$
16.     **return**
```

Figure 5.5: The compute happens-before function

$0 - 7$ respectively. Note that the transition between $0a$ and the fork and the transition between join and $6a$ are also assigned an id. It will also assign the transitions between the fork node and the join node in thread M_2 the $\rho = 1$ and the η from $0-5$ respectively. The set *setHB* will contain the two elements $((0, 0), (1, 0))$ and $((1, 5), (0, 7))$. Note that there is no strict happens-before relation between the transitions in the two threads, between the fork and join nodes.

5.6.2 Lock-Unlock Analysis

We model a lock variable as a semaphore lk. The semaphore lk is represented as a global integer variable initialized to one. The statement lock(lk) acquires the semaphore when $(lk > 0)$ and decreases lk by one, while unlock(lk) releases the semaphore and increases lk by one.

We use static analysis to compute the mutual exclusion due to lock-unlock pat-

```
17. function IsHB($t_1 \in \mathcal{T},\ t_2 \in \mathcal{T}$) : Boolean
18.     let $(\rho_1, \eta_1) = t_1.(\rho, \eta)$
19.     let $(\rho_2, \eta_2) = t_2.(\rho, \eta)$
20.     if $\rho_1 = \rho_2$ then
21.         return $\eta_1 < \eta_2$
22.     for each $((\rho, \eta), (\rho', \eta')) \in setHB$ do
23.         if $(\rho = \rho_1) \wedge (\eta_1 \leq \eta) \wedge (\rho' = \rho_2) \wedge (\eta' \leq \eta_2)$ then
24.             return true
25.     return false
```

Figure 5.6: This function checks if transition t_1 happens-before t_2

tern. We compute the set $\mathcal{O}$ as defined below using a simple conservative algorithm (not shown here).

$$
\begin{aligned}
(lk, t, t') \in \mathcal{O} \quad \text{iff} \quad & t = (gl, i_0, gl_1) \ \wedge\ t' = (gl_n, i_n, gl') \ \wedge \\
& gl \xrightarrow{i_0} gl_1 \xrightarrow{i_1} \cdots gl_n \xrightarrow{i_n} gl' \ \wedge\ i_0 = \mathsf{lock}(lk) \ \wedge\ i_n = \mathsf{unlock}(lk) \\
& \wedge\ \forall_{k \in \{1 \ldots (n-1)\}}\ i_k \notin \{\mathsf{lock}(lk), \mathsf{unlock}(lk)\}
\end{aligned}
$$

Using $\mathcal{O}$ we then update the set $\mathcal{C}_{ij}$ as follows:

$$\mathcal{C}_{ij} = \mathcal{C}_{ij} \setminus \{(t, t') \in \mathcal{C}_{ij} \mid \exists (lk, t_1, t_1'), (lk, t_2, t_2') \in \mathcal{O}.\ t_1 \preceq_{po} t \preceq_{po} t_1' \ \wedge\ t_2 \preceq_{po} t' \preceq_{po} t_2'\}$$

5.7 MAT Analysis

Notation Shortcuts: Before we get into details, we make some notation abuse for ease of readability. Let $t(e)$ represent the transition of the thread where the corresponding event e occurs. When there is no ambiguity, we use an edge to represent a transition. We use e_i to also indicate $t(e_i)$, the transition of thread M_i where the access event e_i belongs. Further, we use $+e_i$ to denote the event immediately after e_i in program order, i.e., $t(+e_i) = next(t(e_i))$. Similarly, we use $-e_i$ to denote event immediately preceding e_i, i.e., $t(e_i) = next(t(-e_i))$. We sometimes refer tuple (a, b) as a pair.

We provide a simple algorithm, GenMAT (Figure 5.7) for generating $\mathcal{MAT}_{ij}$, given a pair of threads M_i and M_j and dependency relation $\mathcal{D}$. We first initialize a queue

```
26. function $\mathsf{GenMAT}(\mathcal{D})$ : $\mathit{SetOfMATs}$
27.     let $\mathcal{MAT}_{ij} := \emptyset$
28.     let $\mathit{worklist}$ := new worklist of $\mathit{Transition} \times \mathit{Transition}$
29.     let $\mathit{done}$ := new queue of $\mathit{Transition} \times \mathit{Transition}$
30.     $\mathit{worklist}.\mathsf{Add}(\vdash_i, \vdash_j)$
31.     while $\mathit{worklist}$ not empty do
32.         let $(f_i, f_j) := \mathit{worklist}.\mathsf{Remove}$
33.         $\mathit{done}.\mathsf{Add}(f_i, f_j)$
34.         let $\mathcal{M}_c := \{m \mid m = (tr_i = f_i \cdots l_i,\ tr_j = f_j \cdots l_j)$ is MAT w.r.t $\mathcal{D}$. $\}$
35.         for each $m \in \mathcal{M}_c$ s.t $\forall m' \in \mathcal{M}_c\ m' \neq m \ \wedge\ l_j \preceq_{po} l'_j$ do
36.                                 /* i.e., $M_j$ has higher priority */
37.             let $\mathcal{MAT}_{ij} := \mathcal{MAT}_{ij} \cup \{m\}$
38.             if $l_i = \dashv_i \ \wedge\ l_j = \dashv_j$ then
39.                 continue
40.             elseif $l_i = \dashv_i$ then
41.                 let $Q := \{(f_i, +l_j)\}$
42.             elseif $l_j = \dashv_j$ then
43.                 let $Q := \{(+l_i, f_j)\}$
44.             else
45.                 let $Q := \{(+l_i, +l_j), (+l_i, f_j), (f_i, +l_j)\}$
46.             let $\mathit{worklist} := \mathit{worklist} \cup Q \backslash \mathit{done}$
47.     return $\mathcal{MAT}_{ij}$
```

Figure 5.7: The GenMAT Algorithm to obtain a set of MATs

worklist with transition pair $(\vdash_i, \vdash_j)$ representing the beginning of the threads (line 30). Next, while the *worklist* is not empty (line 31), for any pair (f_i, f_j) in the *worklist* (line 32), representing the current transitions, we obtain a set of MAT candidates $\mathcal{M}_c$ (line 34) just by using the definition of MAT (Definition 18) for the given dependency relation $\mathcal{D}$. Note that in general there may be many possible MAT starting from a pair (f_i, f_j), but if they are in the same path, we select only one of them such that condition at line 35) is satisfied. In other words, we give M_j the priority over M_i. Note, the choice

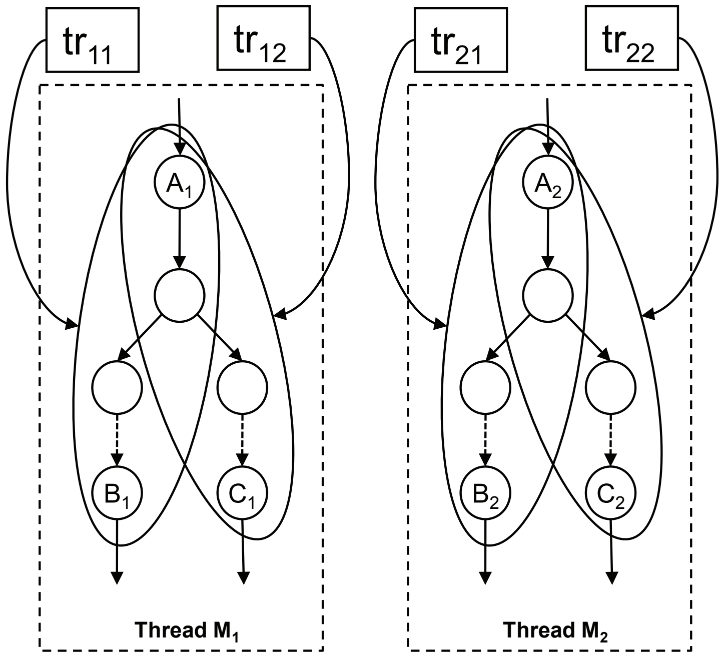


Figure 5.8: MATs for branches

of M_j over M_i is arbitrary, but is required for optimality result. For example, out of the two choices m and m' in Figure 5.3, we prefer m' if we give M_2 a higher priority over M_1.

Furthermore, in a more general setting with conditional branching, we identify MATs by exploring beyond branches, as illustrated in Figure 5.8. Starting from (A_1, A_2), we have following control path segments, $tr_{11} = A_1 \cdots B_1$, $tr_{12} = A_1 \cdots C_1$, $tr_{21} = A_2 \cdots B_2$, and $tr_{22} = A_2 \cdots C_2$ (shown as ovals). For each of the four combinations of tr_{1i}, tr_{2j}, we define MAT separately. Also, note our algorithm do not need to consider loops as the input model is already unrolled.

Due to branches even after pruning of the set $\mathcal{M}_c$ using priority, we may still have more than one MATs in it. For each such MAT m, we update $\mathcal{MAT}_{ij}$ with m and then selectively update the *worklist* as shown in the algorithm (lines 38-46).

Theorem 2. *The algorithm* GenMAT *terminates.*

Proof. For bounded depth, number of pair-wise accesses are bounded. As each transition pair is picked only once (line 32), the procedure terminates. □.

Table 5.1: Run of algorithm GenMAT on example in Figure 5.1

Iter	$(\mathbf{f_i}, \mathbf{f_j})$	$\mathcal{MAT}_{\mathbf{12}}$	*worklist*
0			**((1a, 2a), (1b, 2b))**
1	((1a, 2a), (1b, 2b))	$(1a \cdots 4a, 1b \cdots 3b)$	**((4a, 5a), (1b, 2b))**; **((1a, 2a), (3b, 4b))**; **((4a, 5a), (3b, 4b))**
2	((4a, 5a), (1b, 2b))	$(4a \cdots 5a, 1b \cdots 5b)$	((1a, 2a), (3b, 4b)); ((4a, 5a), (3b, 4b))
3	((1a, 2a), (3b, 4b))	$(1a \cdots 2a, 3b \cdots 5b)$	((4a, 5a), (3b, 4b)); **((2a, 3a), (3b, 4b))**
4	((4a, 5a), (3b, 4b))	$(4a \cdots 5a, 3b \cdots 5b)$	((2a, 3a), (3b, 4b))
5	((2a, 3a), (3b, 4b))	$(2a \cdots 5a, 3b \cdots 5b)$	$\emptyset$

For a given MAT $m = (f_i \cdots l_i, f_j \cdots l_j)$, we define a set of interleaving ordered pairs, $TP(m) = \{(l_i, f_j)), (l_j, f_i))\}$. Given a set of $\mathcal{MAT}_{ij}$, we define $TP(\mathcal{MAT}_{ij}) = \bigcup_{m \in \mathcal{MAT}_{ij}} TP(m)$, and denote it as TP_{ij}. We then define a set TP^- as $(\bigcup_{i \neq j} TP_{ij})$. Intuitively, the set TP^- captures succinctly the token-passing constraints.

Unfortunately, for more than two threads the set TP^- generated using the procedure GenMAT may not be adequate i.e., may miss interleavings. This is due to interference between a MAT and other threads. To overcome this scenario, we define a set of extra token-passing pairs set eTP_{ij} using $\mathcal{MAT}_{ij}$.

$$\begin{aligned} eTP_{ij} = \{ \quad & (l_i, +m_j), (l_j, +m_i) \quad | \quad (f_i \cdots l_i, f_j \cdots l_j) \in \mathcal{MAT}_{ij}, \\ & (f_i \preceq m_i \prec l_i) \wedge \exists_{k \neq j} c_k.(m_i, c_k) \in TP_{ik} \wedge \neg \exists c_j.(m_i, c_j) \in TP_{ij}, \\ & (f_j \preceq m_j \prec l_j) \wedge \exists_{k \neq i} c_k.(m_j, c_k) \in TP_{jk} \wedge \neg \exists c_i.(m_j, c_i) \in TP_{ij} \quad \} \end{aligned}$$

For every pair of threads M_i and M_j, if $Sh_{ij} \subsetneq Sh_i \cup Sh_j$, we generate the set eTP_{ij} from $\mathcal{MAT}_{ij}$. Finally, we construct the set TP as follows:

$$TP = (\bigcup_{i \neq j} TP_{ij}) \cup (\bigcup_{i \neq j} eTP_{ij})$$

Example: We show a run of GenMAT in Table 5.1 for the example in Figure 5.1. We gave M_2 higher priority over M_1. Note that out of the two MATs $m = (tr_1, tr_2)$ and

$m' = (tr'_1, tr'_2)$ (Figure 5.3(a),(b)), *GenMAT* selects m as $2b \prec_{po} 4b$. The table columns provide each iteration step (Iter), the current pair (f_i, f_j) selected, the chosen MAT, and the current *worklist* where, the new pairs added to it are shown in bold.

5.8 Annotation and Path-Balancing

In this component we perform few source-to-source transformations, so that we can use an uniform algorithm to build the model. In the first step, we introduce a global boolean variable token (`tk`) and a clock vector variable (`cs`) in the system. The purpose of these variables are defined below:

- *Token:* Intuitively, the thread with the token variable asserted can execute the current enabled transition. Initially, only one thread, chosen non-deterministically is allowed to assert `tk` and then it is passed around such that at any time only one thread asserts the token.
- *Clock vector:* To model passage of time and thereby to obtain a total ordering on token passing events, we use the concept of Lamport's clock vector [Lam79]. The variable `cs` is a N-tuple $(cs_1, \dots, cs_N)$ and is initialized to 0. Whenever, the token is acquired by a thread T_i, it increments the clock cs_i corresponding to this thread by 1. Intuitively, the clock vector corresponding to the thread with the token captures the number of times the token has been acquired by each thread.

Next, each thread is decoupled from the other threads by *localizing* all the shared variables. This is done by introducing new local variables for each shared variable $v \in \mathcal{V}$ and then substituting every access of v with the new local variable. For clarity, each shared variable $v \in \mathcal{V}$ is represented using the local variable v_{tid}, where *tid* is the id of the thread to which the variable belongs.

In the next steps, we perform thread independent tasks such as node-annotation and path-balancing. We define the set $\Gamma = \{t \mid \exists t'.(t,t') \in TP \vee (t',t) \in TP\}$ as the set of transitions in TP. For each $t \in \Gamma$, we then add two new locations in the transition diagram, a *pre-access location* (`pre`) is inserted before t and a *post-access location* (`post`) is inserted after t. Furthermore, we add instructions in the transitions corresponding to these locations, in particular, for the `pre` location we add that each localized shared variables get a non-deterministic value, while for the `post` location only `tk` gets non-deterministic value. These instrumentations are partly shown in the Figure 5.2. For each transition with a fork instruction we also add a pre-access location called `fs` and add the instruction that the token `tk` gets non-deterministic value. Similarly, for each transition with a join instruction we add a pre-access location called `js` and add instructions that the localized shared variables get non-deterministic values. By adding these extra instructions, our model now have additional behaviors, hence, it is imprecise. In Section 5.9.1 we discuss our representation of each of these thread models using quantifier-free first order logic and then in Section 5.9.2 we describe the constraints that will make these models precise.

We also do path balancing [GG06] in each of the threads. The main idea behind path balancing is to insert dummy *nop* locations and transitions in the transition diagram such that the two sides of the branch have same depth. The purpose of this step is two fold, it allows us to use a simpler uniform algorithm for thread model generation and it also reduces the size of the model. Since, the threads are unrolled, we can use a simple bottom-up traversal algorithm to perform this step.

5.9 Generating Verification Conditions

Given a concurrent program $\mathcal{CS}$, we derive a set of thread models and symbolically check all its schedules for property violations up to a given bounded depth (say D). For this, we create a formula $\Phi_{\mathcal{CS}}$ called *verification condition* such that $\Phi_{\mathcal{CS}}$ is satisfiable iff there exists a schedule in $\mathcal{CS}$ that violates the property. In particular, we use an encoding that creates the formula in a quantifier-free first-order logic to facilitate the application of off-the-shelf SMT solvers as shown in Figure 5.4. The formula $\Phi_{\mathcal{CS}}$ can be further subdivided into the following smaller formulas:

$$\Phi_{\mathcal{CS}} \quad = \quad \Phi_{\mathcal{TM}} \quad \wedge \quad \Phi_{\mathcal{TPM}} \quad \wedge \quad \Phi_{\mathcal{CB}} \quad \wedge \quad \neg\Phi_{\mathcal{PRP}}$$

where, $\Phi_{\mathcal{TM}}$ is the formula corresponding to the thread models, $\Phi_{\mathcal{TPM}}$ is the formula corresponding to the token-passing constraints that captures all possible behaviors, optionally we also add context bounding constraints ($\Phi_{\mathcal{CB}}$), and $\Phi_{\mathcal{PRP}}$ is the formula corresponding to the correctness property that we want to check. In the following sections, we discuss each of this constraints in details.

5.9.1 Thread Models

In this section, we discuss our representation of each of the thread models using quantifier-free first order logic. Recall each thread model $M_i (1 \le i \le N)$ is represented using a transition diagram $\pi_i = (\mathcal{L}_i, \mathcal{I}_i, \rightarrow_i, \iota_i)$. For each transition diagram we perform a control state reachability (CSR) analysis. CSR is a breadth-first traversal of the transition diagram (also can be seen as a CCFG). For a given M_i, we define $R_i(d), 0 \le d \le D$ to be the set of locations statically reachable at depth d.

$$R_i(d) = \{gl' \in \mathcal{L}_i \quad | \quad \iota_i \xrightarrow{i_1}_i gl_1 \cdots \xrightarrow{i_d}_i gl'\}$$

We also use v_i^d to denote the expression for the variable v in the unrolled model M_i at depth d. We define a boolean predicate $B_{gl_i}^d \equiv (PC_i^d = gl_i)$, where PC_i^d is the program counter that tracks the current location in the thread M_i and $gl_i \in \mathcal{L}_i$.

We represent the control relation of a thread M_i using the following constraints. For each transition $(gl, (c, f), gl') \in\rightarrow_i$ such that $gl \in R_i(k-1)$ and $gl' \in R_i(k)$ we add:

$$B_{gl'}^k \Longleftrightarrow c \ \wedge \ B_{gl}^{k-1}$$

We initialize the transition relation by asserting for each thread M_i that $B_{\iota_i}^0$ is true.

Furthermore, we encode the data relation using similar constraints. For each transition $(gl, (c, f), gl') \in\rightarrow_i$ such that $gl \in R_i(k-1)$ and $gl' \in R_i(k)$ and f is an assignment of the form $v := e$ we add:

$$v^k = B_{gl}^k \ ? \ e^{k-1} : v^{k-1}$$

where, e^{k-1} is the expression such that every variable u in e is replaced with variable u^{k-1}. And, for all other variables $x \in V_i, x \neq v$ at depth k we add $x^k = x^{k-1}$

We use the CSR analysis to further reduce the size of the verification condition and also the number of new variables introduced for a depth. Intuitively, for all the variables v that are not written at a depth d, we hash the expression representation for v^d to the existing hash expression for v^{d-1}. This hashing, i.e., reusing of expressions reduces the size of the formula significantly [GG06].

5.9.2 Token-passing Model using MAT

In this section we describe the constraints that makes our model of $\mathcal{CS}$ sound (i.e., it does not miss any necessary interleaving) and complete (i.e., it allows only sequentially consistent traces). Previous approach [GG08] does this by adding token-passing constraints between all pairs of shared accesses. However, our approach reduces such constraints by adding token-passing constraints only between MATs. We exploit the pair-wise atomicity of MATs in a token-based model as follows: We compute the set TP as described in Section 5.7. For each ordered pair $(a, b) \in TP$, we *only* add token-passing constraints from a to b. Recall, such constraints are added between the corresponding `pre` and `post` nodes as discussed in Section 5.3.2. In general, transactions associated with different MATs may not be atomic. For example, tr_1 is not atomic with tr_2'' (Figure 5.3). The following theorem summarizes the adequacy and optimality of the token-passing model using MAT.

Theorem 3 (Ganai et al., 2009 [GK09])**.** *The token-passing model with MATs is adequate i.e., does not miss interleavings. Furthermore, given a dependency relation $\mathcal{D}$, the set TP is optimal i.e., it does not allow two equivalent schedules.*

Next, we describe the various token-passing constraints added in each verification condition in some details. Some of these constraints were first introduced in the token-passing approach [GG08]. We restate them here for completeness of the description. These constraints capture the inter- and intra- thread dependencies due to interleavings, and thereby, eliminate the imprecision introduced in the model by allowing non-deterministic values for the shared variables, up to a bounded depth. The constraints can be divided in four kinds: global constraints, inter-thread constraints, intra-thread constraints, and fork-join constraints,.

Global Constraints

We initialize the token-passing model by allowing *exactly* one thread to have the token. This is captured by the following constraint:

$$(\bigvee_{1 \le i \le N} tk_i^0) \wedge (\bigwedge_{i \neq j} tk_i^0 \Rightarrow \neg tk_j^0)$$

Inter-Thread Constraints

In this subsection we discuss the constraints that are required to capture the token passing from one thread to another. Our goal for the model is to capture every feasible schedule in the concurrent system $\mathcal{CS}$. For this, we introduce few constraints as follows:

Token-Passing Enabling Constraint: For every pair of `pre` locations $gl_i \in R_i(k)$ in thread M_i and `post` locations $gl_j \in R_j(h)$ in thread $M_j, j \neq i$, we introduce a boolean variable rw_{ij}^{kh}, and add the following constraints:

$$rw_{ij}^{kh} \Longrightarrow (B_{gl_i}^{k} \ \wedge \ \neg tk_i^k \ \wedge \ B_{gl_j}^{h} \ \wedge \ tk_j^h \ \wedge \ cs_{ii}^k = cs_{ij}^h)$$

Intuitively, this constraint captures when a token can be passed between threads. More specifically, the token can be passed from the `post` location gl_j at depth h of thread M_j to `pre` location gl_i at depth k of thread M_i if the following holds: (a) The thread M_i is in location gl_i at depth k and does not hold the token, (b) The thread M_j is in location gl_j at depth h and holds the token, and (c) The clock variable cs_i is same in both the threads. If the above condition holds then we say that the token-passing condition is *enabled*. Note that at any point the token-passing condition may be enabled for more than one thread and as such we use the following exclusivity constraints.

Token-Passing Exclusivity Constraint: For every `pre` locations $gl_i \in R_i(k)$, we define a set $rs_i^k = \{rw_{ij}^{kh} \mid i \neq j, 0 \leq h \leq d\}$ which represent the number of locations that can send a token to gl_i. To allow at most one `post` location from another thread to match (pair) with this `pre` location, we assign an unique $id, (0 < id \leq |rs_i^k|)$ to each element of rs_i^k. We also introduce a new variable rc_i^k for each gl_i and require it to take non-zero value iff $rw_{ij}^{kh} = 1$. Similarly, we define the set ws_j^h and the variable wc_j^h for every `post` location in $gl_j \in R_j(h)$.

$$(rw_{ij}^{kh} \Longleftrightarrow rc_i^k \neq 0) \ \wedge \ (0 \leq rc_i^k \leq |rs_i^k|)$$
$$(rw_{ij}^{kh} \Longleftrightarrow wc_j^h \neq 0) \ \wedge \ (0 \leq wc_j^h \leq |ws_j^h|)$$

Intuitively, these exclusivity constraints makes sure that if a token-passing event occurs between a pair of `pre` and `post` locations then all other pairs are implied invalid. These constraints along with the enabling constraints defines rw_{ij}^{kh}. We say a token passing event is triggered iff $rw_{ij}^{kh} = 1$.

Token-Passing Update Constraint: Once the token-passing event is triggered (i.e. $rw_{ij}^{kh} = 1$), then we have to update the state of the thread M_i with the state of the thread M_j. For this, each localized shared variables at $gl_i \in R_i(k+1)$ is updated with the current state values of the localized shared variables at $gl_j \in R_j(h)$. We also assert the token in M_i and deny it in M_j, to indicate the passing of token. The clock variable corresponding to M_i is incremented by one, while the remaining clock variables are synced with that of thread M_j. Thus, for every rw_{ij}^{kh} variable introduced we add the following update constraints:

$$rw_{ij}^{kh} \Longrightarrow (\bigwedge_{p=1}^{m} g_{pi}^{k+1} = g_{pj}^{h}) \wedge (tk_i^{k+1} \wedge \neg tk_j^{h+1}) \wedge (cs_{ii}^{k+1} = cs_{ii}^{k}+1) \wedge (\bigwedge_{q=1, q\neq i}^{N} cs_{qi}^{k+1} = cs_{qj}^{h})$$

Intra-Thread Constraints

In this subsection we discuss the constraints that are required to capture the scenario when none of the token-passing events are triggered for a location in a thread (i.e. no context-switches).

No Token-Passing Update Constraint: For every `pre` locations $gl_i \in R_i(k)$, if none of the token-passing events are triggered for gl_i then the next state values should be unchanged for each localized shared variable in M_i as encoded using the following formula.

$$rc_i^k = 0 \Longrightarrow (\bigwedge_{p=1}^{m} g_{pi}^{k+1} = g_{pi}^{k}) \;\wedge\; (tk_i^{k+1} = tk_i^{k}) \;\wedge\; (\bigwedge_{q=1}^{N} cs_{qi}^{k+1} = cs_{qi}^{k})$$

Similarly, for every `post` locations $gl_j \in R_j(h)$, the next state values should be unchanged for the tk variable.

$$wc_j^h = 0 \Longrightarrow (tk_j^{h+1} = tk_j^{h})$$

Note that we have to add the above constraints because in Section 5.8 we have introduced imprecision in our model by allowing the localized shared variables to have non-deterministic values. These constraints along with the inter-thread constraints makes our model precise.

Write Commit Constraint: To make our model sequentially consistent we want every shared variable write operation to be visible to all other threads. To achieve this our model allows a thread to write to a shared variable only when it has the token. For this, we add for every `post` locations $gl_j \in R_j(h)$ corresponding to only write operations the following constraint:

$$B_{gl_j}^{h} \Longrightarrow tk_j^{h}$$

Fork-Join Constraints

To model the fork instruction we have to make sure that each newly created thread gets the values of the shared variables from the parent thread, and that among the possible threads the token is with exactly one thread. Let $\mathcal{F}$ be the set of thread ids of the newly created threads. For each pair of `fs` location $gl_i \in R_i(k)$ in thread M_i at depth k and the start location of the newly created threads $M_j, j \in \mathcal{F}$, we add the following constraints:

$$(\bigwedge_{p=1}^{m} g_{pi}^{k} = g_{pj}^{0}) \ \wedge \ (\bigwedge_{q=1}^{N} cs_{qi}^{k} = cs_{qj}^{0})$$

Furthermore, if the thread M_i had the token then after fork exactly one of parent thread and the newly created thread has the token as shown below:

$$\begin{aligned} tk_i^k \Longleftrightarrow & (((\textstyle\bigvee_{j\in\mathcal{F}} tk_j^0) \vee tk_i^{k+1}) \wedge \\ & (\textstyle\bigwedge_{j\in\mathcal{F}}(tk_i^{k+1} \Rightarrow \neg tk_j^0) \wedge (tk_j^0 \Rightarrow \neg tk_i^{k+1})) \ \wedge \ (\textstyle\bigwedge_{p,q\in\mathcal{F},p\neq q} tk_p^0 \Rightarrow \neg tk_q^0)) \end{aligned}$$

We treat a `js` location as a slightly modified version of a `pre` location. Let $\mathcal{J}$ be the set of thread ids of the joined threads. For every pair of `js` location $gl_i \in R_i(k)$ and end location $gl_j \in R_j(h), j \in \mathcal{J}$ we define a new boolean variable rw_{ij}^{kh}, and add constraints similar to *token-passing enabling constraint.* Similarly, we also add the *token-passing exclusivity constraint* corresponding to the `pre` location. The *token-passing update constraint* is slightly modified as shown below:

$$rw_{ij}^{kh} \Longrightarrow (\bigwedge_{p=1}^{m} g_{pi}^{k+1} = g_{pj}^{h}) \ \wedge \ tk_i^{k+1} \ \wedge \ (\bigwedge_{q=1}^{N} cs_{qi}^{k+1} = cs_{qj}^{h})$$

We also add the *no token-passing update constraint* corresponding to the `pre` location for the `js` locations. We add a program order constraint between the end location of each joined threads and the `js` location to capture the happens-before relation between them.

Example: We show in Figure 5.9 the token-passing model using MAT for our example in Figure 5.1. We use the set $TP(\mathcal{MAT}_{12})$ generated from the set $\mathcal{MAT}_{12}$ as shown in Table 5.1. We add token-passing constraints (shown as directed edges) in the figure between every ordered pair in the set $TP(\mathcal{MAT}_{12})$. Total number of pair-wise constraints we add is 8, much less compared with all pair-wise constraints (shown in Figure 5.2). The fork/join constraints, shown as dotted edges, provide happens-before ordering between the accesses.

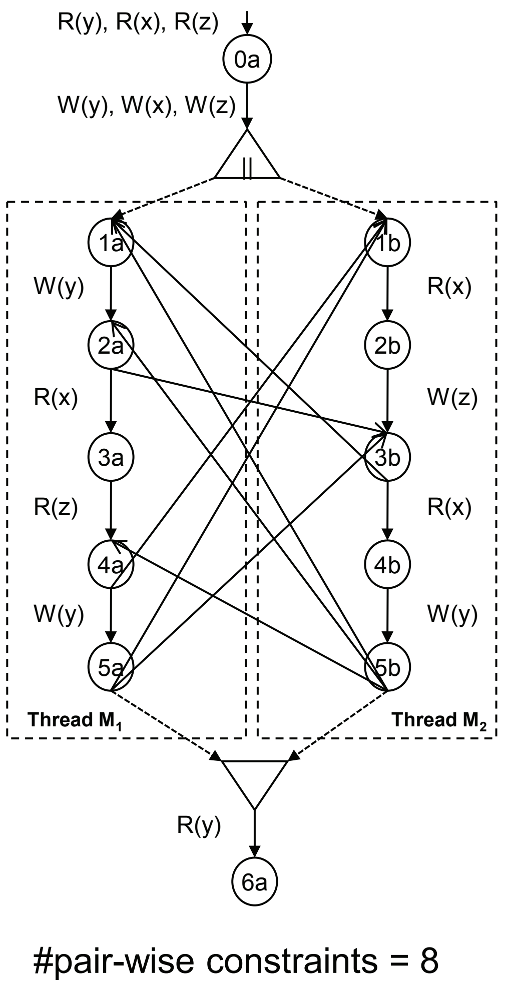


Figure 5.9: The token-passing model using MAT for our running example

5.9.3 Context Bounding

In this section we present a symbolic encoding for the token-based approach that effectively bounds the number of context switches allowed by an interleaving. We add the following constraints to bound the number of times a token could be passed to a specific thread model M_i.

$$CB_i^l \leq \texttt{cs}_{ji} \leq CB_i^u, \quad 1 \leq j \leq N$$

where, CB_i^l and CB_i^u are user-provided lower and upper context-bounds respectively. In our approach, we allow thread specific fine-grained context bounding. With this flexivity we can now have different context bounds for different threads. These constraints are

Table 5.2: Details of the benchmarks checked during our experiments

Ex	#SA	$\#C^-$	$\#C^+$	$\#TP$	$\#TP_U$	$\#TP_M$	#M	#SV	#T
E1	(3,3,1)	8	1	30	6	4	2	2	15
E2	(3,4,4)	20	7	80	32	18	10	2	28
E3	(3,5,4)	24	9	94	40	23	14	2	32
E4	(2,2,2)	8	2	24	8	6	3	1	15
E5	(1,4,4)	12	8	48	32	23	14	2	19
E6	(2,8,8)	36	4	192	8	8	4	3	29
E7	(1,20,20)	220	200	880	800	591	390	2	51
E8	(2,40,40)	660	100	3520	200	200	100	3	93
E9	(1,100,100)	5100	5000	20400	20000	14951	9950	2	211
E10	(2,200,200)	15300	2500	81600	5000	5000	2500	3	413

#SA - No. of Shared Accesses in each thread. #T - No. of Transitions.
#M - No. of MATs. #SV - No. of Shared Variables.
$\#C^-$ and $\#C^+$ - No. of Conflict before and after UA.
$\#TP$ - No. of Token-Passing (TP) constraints in basic encoding [GG08].
$\#TP_U$ - No. of TP constraints using UA.
$\#TP_M$ - No. of TP constraints using MATs.

optional and are not required for the correctness of the model. In general, by adding these constraints we make our approach incomplete, however, recent findings show that in practice concurrency bugs can often be exposed in interleavings with a surprisingly small number of context switches [QR05, LR08, Mus08].

Finally, we generate verification conditions comprising of the transition relation of each thread model, token-passing constraints, context-bounding constraints (optionally), and negated property constraints. These constraints are then expressed in a quantifier-free formula and passed to a SMT solver for a satisfiability check as shown in Figure 5.4.

Table 5.3: Timing results of the benchmarks checked during our experiments

Benchmarks	B [GG08]			B+M			Ext [WYKG08]
(S/U)	NCB	C1	C2	NCB	C1	C2	NCB
1. E1S	00.01	00.01	00.01	00.01	00.01	00.01	00.02
2. E2S	00.03	00.04	00.04	00.02	00.02	00.03	00.04
3. E3S	00.03	00.04	00.04	00.03	00.03	00.02	00.06
4. E4S	00.01	00.01	00.02	00.01	00.01	00.01	00.01
5. E4U	00.01	00.01	00.01	00.01	00.01	00.01	00.01
6. E5S	00.06	00.03	00.07	00.02	00.02	00.03	00.04
7. E6S	00.18	00.12	00.13	00.01	00.02	00.01	00.24
8. E7S	22.10	00.90	01.10	07.29	00.54	00.68	00.84
9. E8S	1550.56	08.50	61.92	00.17	00.16	00.19	04.18
10. E8U	TO	13.72	413.48	50.54	00.16	01.30	99.71
11. E9S	TO	497.81	1742.57	TO	150.81	515.95	MO
12. E10S	TO	TO	TO	TO	16.44	20.57	MO
B - Basic token-based approach [GG08]. M - Using MAT. Ext [WYKG08].							
NCB: No context bound C1: One context bound. C2: Two context bound.							
MO: Memory out. TO: Time out. Time is in secs.							

5.10 Experiments and Results

We implemented our approach in a token-based symbolic tool called Candor similar to [GG08], using the SMT solver Yices-1.0.13 [SRI]. Figure 5.4 presents an overview of Candor. The implementation of the Unreachability analysis (UA) and the MAT analysis follows closely the algorithm described in Section 5.6 and Section 5.7 respectively. It then incrementally adds concurrency constraints at the current depth k using the MAT table and the context bound information. The resultant verification condition is then given to the SMT solver. Next, depending on the result of the SMT solver and the current depth, our algorithm either produces a counterexample or repeats the above steps again or aborts.

In our experiments, we automatically checked several benchmarks of varied complexity with respect to the number of shared variable accesses. These benchmarks correspond to two or three-threaded systems. We obtained these benchmarks by combining several trace programs each corresponding to a concurrent run of two-threaded concurrent system. We also replaced fixed program input values in the trace program by symbolic values. The property constraints correspond to assertion violations. All benchmarks are checked at a depth D equal to the longest path in the program (as it is already unrolled).

The details of the benchmarks that we checked are shown in Table 5.2, along with the number of shared variable accesses in each thread, the number of conflicts before and after UA, the number of pair-wise constraints in the basic encoding and then using UA and then using MATs, the number of MATs, the number of shared variables in the program, and the number of transitions in the program. Note that by using MATs the number of pair-wise constraints are reduced. For example, benchmark E7 has a reachable violation with three threads with 1, 20, and 20, number of shared accesses, respectively. Also, E7 benchmark has 220 and 200 conflicts before and after UA respectively, 390 MATs, 2 shared variables, and 51 transitions.

Table 5.3 presents the timing results of the benchmarks checked by Candor. We conducted our experiments on a Linux box with Intel Core 2 CPU T7200 at 2.0 GHz Processor with 1GB physical memory running Ubuntu Linux 8.04, using a 1800 secs time limit (represented using TO in Table 5.3). Each benchmark is suffixed with S or U corresponding to the satisfiable (i.e., has a reachable violation) or unsatisfiable instance. Column 1 gives the name of the benchmarks. In Columns 2 – 4, we present the results of token-based approach [GG08] using TP constraints, referred to as the encoding `B`. In these columns, we provide the time taken (in secs) with no context-bound constraint (NCB), time taken with one context-bound per thread ($C1$), and time taken with two context-bound per thread ($C2$). In Columns 5 – 7, we present similar results for our approach using MAT analysis, denoted as `B+M`, i.e., token-based approach using TP_M constraints. In the last column, we compare our results with another state-of-the-art symbolic approach [WYKG08] based on synchronous modeling, referred to as `Ext`. Since `Ext` does not support context-bounding, and it is not clear how to add those constraints efficiently, we do not have any reportable data.

The encoding using MAT (`B+M`) significantly outperforms the basic encoding `B`, and `Ext` in timing performance. Encoding using MATs with context bounding (`B+M+C1`

Table 5.4: Size of the formula for the depth D (i.e. entire program)

Benchmarks	B	B+M	Ext
1. E1	16K	11K	73K
2. E2	35K	27K	118K
3. E3	42K	34K	149K
4. E4	16K	13K	34K
5. E5	32K	26K	174K
6. E6	98K	18K	497K
7. E7	487K	375K	1.7M
8. E8	1.9M	163K	6.9M
9. E9	12M	8.7M	51M
10. E10	48M	3.3M	256M

Table 5.5: Depth at which witness was found for the benchmarks checked

Benchmarks	B	B+M	Ext
1. E1	9	9	10
2. E2	12	12	17
3. E3	14	14	19
4. E4	9	9	10
5. E5	11	11	14
6. E6	15	15	24
7. E7	27	27	46
8. E8	47	47	88
9. E9	107	107	206
10. E10	-	207	408

and B+M+C2) can find the SAT instances very quickly, whereas other encoding *cannot* find it within the time limit.

Table 5.4 further compares the size of the B+M encoding with Ext and the B encoding. Each row gives the size of the *formula* for the verification condition corresponding to the bound D (i.e. entire program). Note that this is not the amount of memory used by the SMT solver while checking for a SAT or UNSAT instance. The table shows that the size in `B+M` encoding is significantly smaller (up to 1 - 2 orders of magnitude) than that of `Ext` and `B` encoding. This in turn may allow the SMT-solver to solve bigger instances.

Table 5.5 shows the depth at which the witness was found for all the benchmarks, while checking for SAT instances. Note, due to synchronous modeling, the witness length D tends to be larger for `Ext`, also noted in [GG08].

5.11 Counterexample

The SMT solver generates a counterexample once it finds a satisfiability instance. This counterexample usually consists of a set of values for the variables in our model corresponding to an interleaving in the program such that the assertion (property) we are checking is violated. Unfortunately, the counterexample generated by the solver is usually very hard to debug. In our tool we rebuild the counterexample trace over our CCFG representation of the program. We believe this visual representation makes debugging of bugs easier and faster.

Example: We show in Figure 5.10 a counterexample for our running example from Figure 5.1. The variables x, y, z all gets the value zero which is in accordance to the initial assumption. The trace in the CCFG is shown using solid bold arrows. Each transition has the value of the variable it updates. The context switches between the threads are shown using dashed bold arrows.

5.12 Related Work

In Section 5.1 we discussed various related work for state-space reduction for symbolic approaches. We now give a brief overview, with a few representative related work in combining partial-order reduction method to address state-space explosion problem in explicit approaches.

In *explicit* approaches, typically, the concurrent system is executed with interleaving semantics using a scheduler. In such model checkers [God97, AQR$^+$04], the scheduler is constrained – by adding more constraints, i.e., guard strengthening – to explore only

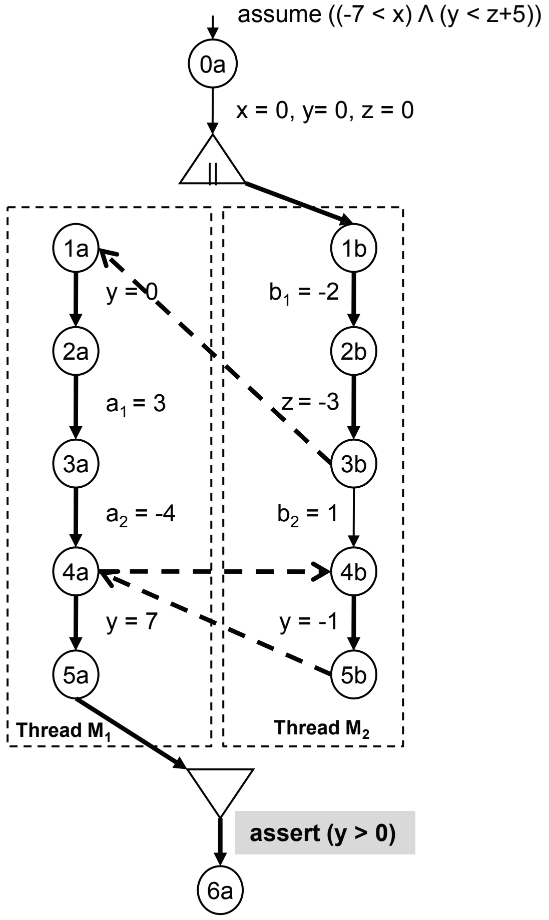


Figure 5.10: A counterexample on the CCFG for our running example

a subset of interleaving that guarantee the correctness. At a given global state, only a subset of transitions (for e.g., persistent set [God95]) are explored. One can obtain the persistent set using conservative static analysis. Since the static analysis does not provide precise dependency relation, one can obtain the set dynamically [FG05]. One can also use conditional dependency relation to declare two transitions being dependent with respect to a given state [GP93]. One can also use sleep set [God95] to eliminate redundant interleaving not eliminated by persistent set. In previous works, researchers have also used lockset-based transactions to cut down interleaving between access points that are provably unreachable [FQ03, SC06, LPQR05]. In spite of these efforts, the scalability problem remains. To overcome this limitation, some researchers have employed sound abstraction [AQR$^+$04] with bounded (i.e., under-approximation) number of context switches [QR05], while some others have used finite-state model abstractions [CKS05], combined with proof-guided method to discover the context switches [GLST05].

There have been parallel efforts [AHMN91, YGL04, BAM07] to detect bugs for weaker memory models. As shown in [YGLS03], one can check these models using axiomatic memory style specifications combined with constraint solvers. We are not aware of any approach combining partial-order reduction with these axiomatic style to reduce constraints.

5.13 Summary

In this chapter we described a bounded depth symbolic analysis tool called Candor for concurrent programs. We are first to exploit partial-order reduction techniques in a symbolic analysis effort that generates verification conditions directly without an explicit scheduler. We discussed a novel approach to reduce verification problem sizes and state space for concurrent systems using MATs. We also leverage our approach using static analysis techniques that identifies simultaneously unreachable conflicting accesses. We also efficiently encode thread-specific context bounding in our approach. Our experimental results demonstrates the efficacy of our approach. Our MAT-based approach is orthogonal to the approaches that exploit transaction-based reductions [SC06, FQ03, KGS06]. Nevertheless, we can exploit those to identify unreachable conflicting pairs, and further reduce the necessary token-passing constraints.

Acknowledgements: Chapter 5, in part, have been accepted for publication as "Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions" by Sudipta Kundu and Malay Ganai in *SPIN 09: Proceedings of the 16th International SPIN Workshop on Model Checking of Software* [GK09]. The dissertation author was the primary investigator and author of the parts of this paper that is included in this dissertation.

Chapter 5, in part, is in preparation for publication as "Efficient Symbolic Analysis for Concurrent Programs" by Sudipta Kundu and Malay Ganai. The dissertation author is the primary investigator and author of this paper.

Chapter 6

Translation Validation of High-Level Synthesis

Once the important properties of the high-level components have been verified possibly using techniques presented in Chapter 4 and 5, the translation from the high-level design to low-level RTL still needs to be proven correct, thereby also guaranteeing that the important properties of the components are preserved. In this chapter we will discuss an approach that proves that the translation from high-level design to a scheduled design is correct, for each translation that the HLS tool performs. In the next chapter we will describe another approach that will allow us to write part of these tools in a provably correct manner.

6.1 Overview of Translation Validation

HLS tools are large and complex software systems, often with hundreds of thousands of lines of code, and as with any software of this scale, they are prone to logical and implementation errors. Apart from applying a monolithic tool, HLS process is characterized by significant user intervention from recoding to directing the synthesis goals. Consequently, the HLS process, even with automated HLS tools, is error prone and may lead to the synthesis of RTL designs with bugs in them, which often have expensive ramifications if they go undetected until after fabrication or large-scale production. Hence, correctness of the HLS process (manual or automatic) has always been an important concern.

In general, proving that the HLS process *always* produces target RTL designs that are semantically equivalent or refinement to their source versions is usually very hard. However, even if one cannot prove the HLS process correct once and for all, one can try to show, for each translation that HLS performs, that the output program produced by these steps has the same behavior as the original program. Although this approach does not guarantee that the HLS process is bug free, it does guarantee that any errors in translation will be caught when the particular steps of HLS are performed, preventing such errors from propagating any further in the hardware fabrication process. This approach to verification, called *translation validation*, has previously been applied with success in the context of optimizing compilers [PSS98, Nec00, RM99, ZPFG03, GZB05].

6.2 Overview of Our Approach

During the HLS process, an engineer starts with a high-level description of the design, usually called a specification, which is then refined into progressively more concrete implementations. Checking correctness of these refinement steps has many benefits, including finding bugs in the translation process, while at the same time guaranteeing that properties checked at higher-levels in the design are preserved through the refinement process, without having to recheck them at lower levels. For example, if one checks that a given specification satisfies a safety property, and that an implementation is a correct trace refinement of the specification, then the implementation will also satisfy the safety property. In this chapter, we show using a novel algorithm how translation validation can effectively be implemented in a previously unexplored setting, namely HLS. The novelty of our approach comes from the fact that it can account for concurrency which is inherent in hardware design. Our algorithm deals with this concurrency using standard techniques for computing weakest preconditions and strongest postconditions of parallel programs [Cha88].

Our translation validation algorithm uses a simulation relation approach to prove refinement. Our algorithm consists of two components. The first component is given a relation, and checks that this relation satisfies the properties required for it to be a correct refinement simulation relation. The second component automatically infers a correct simulation relation just from the specification and the implementation programs. In particular, our inference algorithm automatically establishes a relation that states what points in the implementation program are related to what points in the specification program. This relation guarantees that for each execution sequence in the implementation, an equivalent execution sequence exists in the specification. Apart from refinement checking, we also generalize both of our checking and inference algorithms to prove equivalence between the specification and the implementation programs using a bisimulation relation approach.

To evaluate our approach, we used the Simplify theorem prover [DNS05] to implement our algorithms in a validating system called Surya. We then used Surya to check the correctness of a variety of refinements of infinite state concurrent systems represented using Communicating Sequential Processes [Hoa85] (CSP) programs. Next, we used Surya to validate the results of a realistic HLS tool. In particular, we validate all the phases (except for parsing, binding and code generation) of the Spark HLS tool [GDGN03] against the initial behavioral description. With over 4,000 downloads, and over 100 active members in the user community, Spark is a widely used tool. Although commercial HLS tools exists, these tools are not available for academic experimentation – Spark represents the state of the art in the academic community.

Our verification approach is modular as it works on one procedure at a time. Furthermore, for Spark our validation tool took on average 6 seconds to run per procedure, showing that translation validation of HLS transformations can be fast enough to be practical. Finally, in running Surya, two failed validation runs have lead us to discover two previously unknown bugs in the Spark tool. These bugs cause Spark to generate *incorrect* RTL for a given high-level program. This demonstrates that translation validation of the HLS process can catch bugs that even testing and long-term use may not uncover.

Contributions

Our main contributions can be summarized as follows:

1. We show using a novel algorithm how translation validation can effectively be implemented in a previously unexplored setting, namely HLS.

2. We developed a fully automated algorithm that uses an automated theorem proving component in order to handle infinite state spaces.

3. Our approach can be seen as a generalization of existing translation validation techniques to account for concurrency and for trace refinement checking.

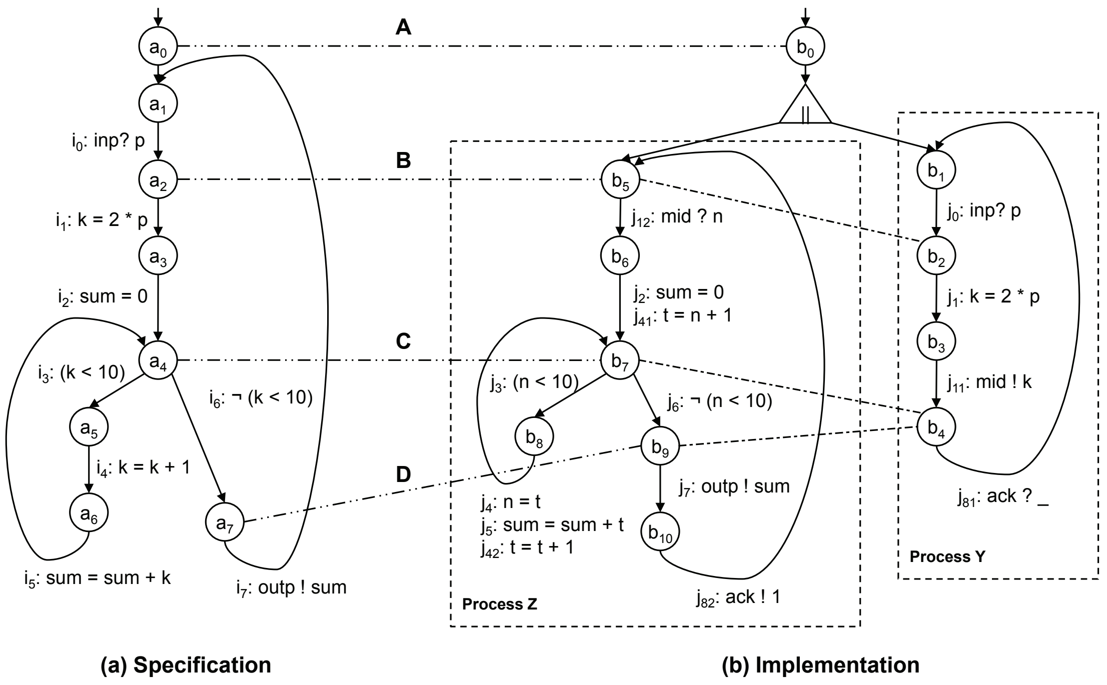


Figure 6.1: CCFGs of our running example along with a simulation relation

4. We implemented our algorithm in a prototype tool called Surya using the Simplify theorem prover.

5. In our experiments, we used Surya to check the correctness of a variety of CSP refinements and also to validate all the phases (except for parsing, binding and code generation) of the Spark HLS tool [GDGN03] against the initial behavioral description.

6.3 Illustrative Example

At the heart of a HLS process is a model of a system consisting of concurrent pieces of functionality, often expressed as sequential program-like behavior, along with synchronous or asynchronous interactions [LSV98, SJ04]. CSP is a calculus for describing such concurrent systems as a set of processes that communicate synchronously over explicitly named channels. In this chapter we describe our algorithm using CSP-style concurrent programs. While CSP presents a good model for a large number of hardware models described using HDLs, we note that the core algorithms of our approach do not

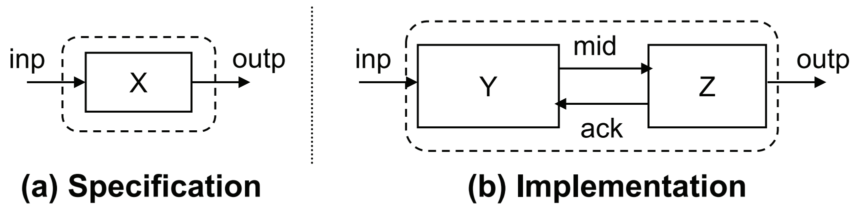


Figure 6.2: Communication diagrams of our running example

depend on the choice of the input language. For example, in our experiments we have used our approach for programs that may include arrays, and function calls that are generally not part of CSP programs.

We start out by describing the salient features of CSP required for understanding the examples in this chapter. A CSP program is a set of (possibly mutually recursive) process definitions. A asynchronous parallel composition of two processes P and Q is written as $(P \mid\mid Q)$. Asynchronous parallel processes in our version of CSP (and Hoare's original version [Hoa85]) can only communicate through messages on channels. Although there are no explicit shared variables, these can easily be simulated using a process that stores the value of the shared variable, and that services reads and writes to the variable using messages. $c?v$ denotes reading a value from a channel c into a variable v and $c!v$ denotes writing a variable v to a channel c. Reads and writes are synchronous. Channels can be visible or hidden. Visible channels are externally observable, and these are the channels that we preserve the behavior of when checking for correctness. We also allow simple C-style control instructions and synchronous parallel composition. By allowing both asynchronous (inherent in CSP) and synchronous semantics of concurrency we support system designs which are *Globally Asynchronous Locally Synchronous* (GALS).

We now present a simple example that illustrates our approach (Figure 6.1). For now ignore the dashed lines in the figure. The specification is a sequential process X shown in Figure 6.1(a) using our internal Concurrent Control Flow Graph (CCFG) representation after tail recursion elimination has been performed. We omit the details of the actual CSP code, because the CCFG representation is complete, and we believe the CSP code only makes the example harder to follow. This process is continually reading values from an input channel called inp into a variable p and then computes the sum from $(2\times\mathsf{p}+1)$ to 10 using a loop. Finally, it writes the sum out to a channel named outp. In refinement based hardware development, the designer often starts with such a

Table 6.1: A simulation relation for our running example

	(gl_1, gl_2, ϕ)
A.	$(a_0, b_0, true)$
B.	$(a_2, (b_2, b_5), \mathsf{p_s} = \mathsf{p_i})$
C.	$(a_4, (b_4, b_7), \mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (\mathsf{k_s} + 1) = \mathsf{t_i})$
D.	$(a_7, (b_4, b_9), \mathsf{sum_s} = \mathsf{sum_i})$

high-level description of a sequential design, refining the details of the implementation later on.

An implementation (Figure 6.1(b)) may use two separate parallel processes (components) Y and Z, communicating via a hidden channel mid and an acknowledgment channel ack as shown in Figure 6.2(b). Like its specification it also takes a value from the inp channel into a variable p and outputs the sum from $(2 \times \mathsf{p} + 1)$ to 10 in the outp channel. However, now it does so in 2 steps, first the process Y multiplies p by 2 and sends it to the component Z then process Z computes the sum and writes it to the outp channel. One additional subtlety of this example is that, in order for the refinement to be correct, an additional channel needs to be added for sending an acknowledgment token (in this case the value 1) back to the process Y, so that a new value isn't read from the inp channel until the current value has been written out to the outp channel. The value read from the ack channel is not used, and so we use an "_" for the variable being read. Instructions on the same transition edge are executed in *parallel* (synchronously).

Apart from the architectural differences, the loop-structure in the implementation is different from the one in the specification in several ways. First, a loop-shifting transformation has moved the operation $\mathsf{i_4}$ from the beginning of the loop body to the end of the loop body ($\mathsf{j_{42}}$), while also placing a copy of the operation in the loop header ($\mathsf{j_{41}}$) using the temporary variable t. The effect of this loop-shifting transformation is a form of software pipelining [Lam88]. Note that without this pipelining transformation it would not have been possible to schedule the operation $\mathsf{i_4}$ and $\mathsf{i_5}$ together due to the data dependence between them. In addition to loop-shifting, a copy propagation of instruction $\mathsf{j_4}$ to $\mathsf{j_5}$ and $\mathsf{j_{42}}$ are also performed. This ability to make large scale code transformations via parallelizing code transformations as shown here is an important aspect of parallelizing HLS implemented in SPARK. Even without HLS tools, similar source-level transformations are often done manually by the designer to optimize the generated code as a part of high-level design process.

Translation Validation Approach

Our translation validation approach consists of two parts, which theoretically are independent, but for practical reasons, we have made one part subsume the other as explained below. The first part is a *checking algorithm* that, given a relation, determines whether or not it satisfies the properties required for it to be a valid simulation relation. The second part is an *inference algorithm* that infers a relation given two programs, one of which is a specification, and the other is an implementation. To check that one program is a refinement to another, one therefore runs the inference algorithm to infer a relation, and then one uses the checking algorithm to verify that the resulting relation is indeed the required relation. Because the inference algorithm does a similar kind of exploration as the checking algorithm, this leads to duplicate work. To reduce this work, we have made the inference algorithm also perform checking, with only a small amount of additional work. This avoids having the checking algorithm duplicate the exploration work done by the inference algorithm. The checking algorithm is nonetheless useful by itself, in case our inference algorithm is not capable of finding an appropriate relation, and the relation is manually provided by the system designer.

Simulation Relation

The goal of the simulation relation in our approach is to guarantee that the specification and the implementation interact in the same way with any surrounding environment that they would be placed in. The simulation relation guarantees that the set of execution sequences of visible instructions in the implementation is a subset of the set of execution sequences in the specification. In what follows, we consider visible instructions to be read and write operations to visible channels. However, in Section 6.8.2, we define visible instructions to be function calls and return statements.

The simulation relation (defined formally in Section 6.5) consists of a set of entries of the form (gl_1, gl_2, ϕ), where gl_1 and gl_2 are program locations in the specification and implementation respectively, and ϕ is a predicate over variables of the specification and implementation. The pair (gl_1, gl_2) captures how the control state of the specification is related to the control state of the implementation, whereas ϕ captures how the data is related. For our running example, the entries in the simulation relation are labeled A through D in Figure 6.1, and each entry has a predicate associated with it as shown in Table 6.1.

The first entry 'A' in the simulation relation relates the start location of the specification and the implementation. For this entry, the relevant data invariant is *true*, as we have no information about the states of the programs in those locations. The second entry 'B' shows the specification just as it finishes reading a value from the inp channel. The corresponding control state of the implementation has the Y process in the same state, just as it finishes reading from the inp channel and the other process Z is at the top of its loop. We use subscript s to denote variables in the specification and subscript i for variables in the implementation. For this entry, the relevant data invariant is $\mathsf{p_s} = \mathsf{p_i}$, which states that the value of p in the specification is equal to the value of p in the implementation. This is because both the specification and the implementation have stored in p the same value from the surrounding environment. In the next subsection Inference Algorithm, we explain in further detail how our algorithm models the environment as a set of separate processes that are running in parallel with the specification and the implementation. For now we hide these additional processes for clarity of exposition.

The next entry 'C' in the simulation relation relates the loop head (a_4) in the specification with the loop head (b_7) of the Z process in the implementation. This entry represent two loops that run in synchrony, one loop being in the specification and the other being in the implementation. The invariant can be seen as a loop invariant across the specification and the implementation, which guarantee that the two loops produce the same effect on the visible instructions. The data part of this entry guarantee that the two loops are in fact synchronized. Nominally, we need at least one entry in the simulation that "cuts through" every loop pair, in the same way that there must be at least one invariant through each loop when reasoning about a single sequential program.

The last entry 'D' in the simulation relation relates the location a_7 in the specification with the location (b_4, b_7) of the implementation. The relevant invariant for this entry is $\mathsf{sum_s} = \mathsf{sum_i}$, since the specification is about to write sum to the externally visible outp channel and the implementation is about to write sum to the same channel (our correctness criterion).

Simultaneous execution from the last entry 'D' can reach back to 'B', establishing the invariant $p_s = p_i$, since by the time execution reaches the second entry again, both the specification and the implementation would have read the next value from the environment (details of how our algorithm establishes that the two next values read from the environment processes are equal is explained in the Inference Algorithm subsection).

Checking Algorithm

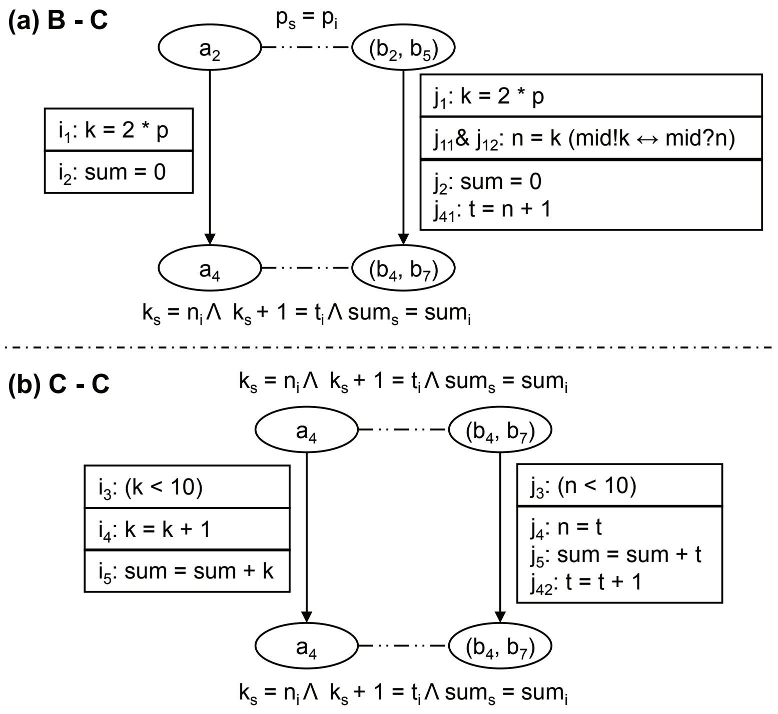


Figure 6.3: Checking the simulation relation. (a) Traces from B to C (b) Traces from C to C

The entries in the simulation relation must satisfy some simple local requirements (which are made precise in Section 6.5). Intuitively, for any entry (gl_1, gl_2, ϕ) in the simulation relation, if the specification and implementation start executing in parallel at control locations gl_1 and gl_2 in states where ϕ holds, and they reach another simulation entry (gl'_1, gl'_2, ϕ'), then ϕ' must hold in the resulting states.

Given a simulation relation, our checking algorithm checks each entry in the relation individually. For each entry (gl_1, gl_2, ϕ), it finds all other entries that are reachable from (gl_1, gl_2), without going through any intermediate entries. For each such entry (gl'_1, gl'_2, ψ), we check using a theorem prover that if (1) ϕ holds at gl_1 and gl_2, (2) the

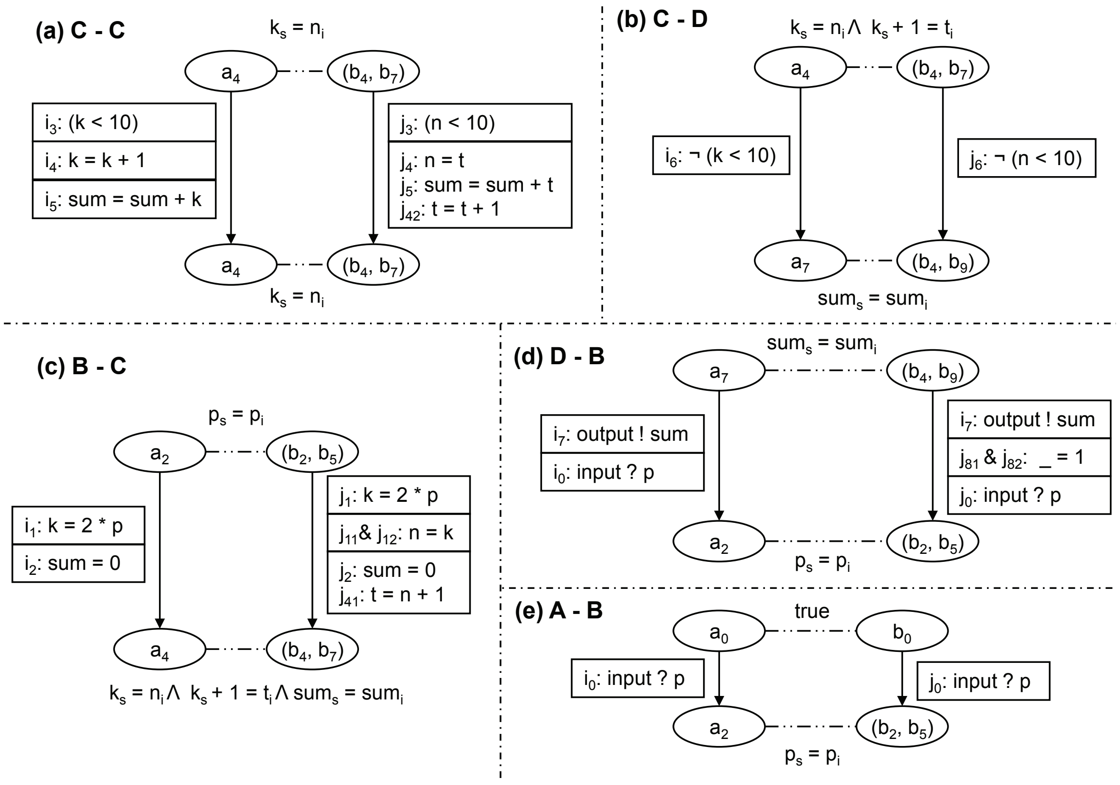


Figure 6.4: Steps of the 2^{nd} iteration for computing the simulation relation

specification executes from gl_1 to gl'_1 and (3) the implementation executes from gl_2 to gl'_2, then ψ will hold at gl'_1 and gl'_2.

For our example, the traces in the implementation and the specification from B to C and the trace from C to itself are shown in Figures 6.3(a) and 6.3(b) respectively. The communication events have been transformed into assignments and the original communication events are in brackets.

For the B - C path shown in Figure 6.3(a), our algorithm uses a theorem prover to validate that if $\mathsf{p_s} = \mathsf{p_i}$ holds before the two traces, then $\mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (\mathsf{k_s}+1) = \mathsf{t_i})$ holds after the traces have been executed. Similarly, it checks the traces from C to C shown in Figure 6.3(b) and also all the other entries in the simulation relation. If there were multiple paths from an entry, our algorithm checks all of them.

Table 6.2: Iterations for computing the simulation relation

(gl_1, gl_2)	1st iteration	2nd iteration	3rd iteration (ϕ)
A. (a_0, b_0)	*true*	*true*	*true*
B. $(a_2, (b_2, b_5))$	$\mathsf{p_s} = \mathsf{p_i}$	$\mathsf{p_s} = \mathsf{p_i}$	$\mathsf{p_s} = \mathsf{p_i}$
C. $(a_4, (b_4, b_7))$	$\mathsf{k_s} = \mathsf{n_i}$	$\mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i}$ $\wedge(\mathsf{k_s} + 1) = \mathsf{t_i}$	$\mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i}$ $\wedge(\mathsf{k_s} + 1) = \mathsf{t_i}$
D. $(a_7, (b_4, b_9))$	$\mathsf{sum_s} = \mathsf{sum_i}$	$\mathsf{sum_s} = \mathsf{sum_i}$	$\mathsf{sum_s} = \mathsf{sum_i}$

Inference Algorithm

Our inference algorithm starts by finding the pairs of locations in the implementation and the specification that need to be related in the simulation. In the given example, our algorithm first adds (a_0, b_0) as a pair of interest, which is the entry location of both programs. Then it moves forward simultaneously in the implementation and the specification until it reaches a branch or an operation (read or write) on a visible channel. In the example from Figure 6.1, our algorithm finds that there is a branch, an input and an output event that must be matched (the specification events inp?p and outp!sum should match, respectively, with the implementation events inp?p and outp!sum). This amounts to computing the first column of Table 6.2. While finding these pairs of locations, our algorithm also does two things. First, it correlates the branch in the specification and the implementation (details of how we establish branch correlations is explained in Section 6.6). Next, it finds the local conditions that must hold for the visible events to match. For events that output to externally visible channels, the local condition states that the written values in the specification and the implementation must be the same. For example, the local condition for the output event is $\mathsf{sum_s} = \mathsf{sum_i}$.

For events that read from externally visible channels, the local condition states that the specification and the implementation are reading from the same point in the conceptual stream of input values. To achieve this, we use an environment process that models each externally visible input channel c as an unbounded array values of input values, with an index variable i stating which value in the array should be read next. This environment process runs an infinite loop that continually outputs values[i] to c and increments i. Assuming that i and j are the index variables from the environment processes that model an externally visible channel c in the specification and the implementation, respectively, then the local condition for matching events c?a (in the specification) and c?b (in the implementation) would then be $i_\mathsf{s} = j_\mathsf{i}$. The equality between the index variables implies that the values being read are the same, and since this fact is always true, we directly add it to the generated local condition, producing $i_\mathsf{s} = j_\mathsf{i} \wedge a_\mathsf{s} = b_\mathsf{i}$.

Once the related pairs of locations have been collected we define, for each pair of locations (gl_1, gl_2), a constraint variable $\psi_{(gl_1,gl_2)}$ to represent the state-relating formula that will be computed in the simulation relation for that pair. We then define a set of constraints over these variables to ensure that the would-be simulation relation is a simulation.

There are two kinds of constraints. First, for each pair of locations (gl_1, gl_2) that are related, we want $\psi_{(gl_1,gl_2)}$ to imply that the local condition at those locations hold. For example, $\psi_{(a_7,(b_4,b_9))}$ should imply $\mathsf{sum}_\mathsf{s} = \mathsf{sum}_\mathsf{i}$, so that the output values are the same. Such constraints guarantee that the computed simulation relation is strong enough to show that the visible instructions behave the same way in the specification and the implementation. A second kind of constraint is used to state the relationship between one pair of related locations and other pairs of related locations. For example, if starting at (gl_1, gl_2) in states satisfying $\psi_{(gl_1,gl_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations (gl'_1, gl'_2), then $\psi_{(gl'_1,gl'_2)}$ must hold in the resulting states. As shown in Section 6.6, such constraints can be stated over the variables $\psi_{(gl_1,gl_2)}$ and $\psi_{(gl'_1,gl'_2)}$ using the weakest precondition operator (wp). This second kind of constraint guarantees that the computed simulation relation is in fact a simulation.

Once the constraints are generated, we solve them using an iterative algorithm that starts with all constraint variables set to *true* and then iteratively strengthens the constraint variables until a theorem prover is able to show that all constraints are satisfied. Although in general this constraint-solving algorithm is not guaranteed to terminate, in practice it can quickly find the required simulation relation.

The constraint solving for our example is shown in Table 6.2. Our algorithm first initializes the constraint variables with the local conditions that are required for the visible instructions to be equivalent. Then it chooses any entry from the table, say C and finds the entries that can reach it (i.e. C and B). Consider the synchronized loop from C to C shown in Figure 6.4(a). Our algorithm computes the weakest precondition of the formula at the bottom ($\mathsf{k_s} = \mathsf{n_i}$) over the instructions in the implementation and in the specification, which happens to be $\delta = [(\mathsf{k_s} < 10) \Rightarrow (\mathsf{n_i} < 10) \Rightarrow (\mathsf{k_s} + 1) = \mathsf{t_i}]$. Next, it asks a theorem prover if the condition at the top i.e. $\mathsf{k_s} = \mathsf{n_i}$ implies δ. Since it does not, our algorithm strengthens the constraint variable at the top with $(\mathsf{k_s} + 1) = \mathsf{t_i}$ which is a stronger condition than δ. A similar pass through Figure 6.4(b) strengthens the constraint variable at C with ($\mathsf{sum_s} = \mathsf{sum_i}$). For the other paths B - C, D - B, and A - B shown in Figure 6.4 the theorem prover is able to validate the implication, and as such we do not need to strengthen. Our constraint solving continues in this manner until a fixpoint is reached.

6.4 Definition of Refinement

We now present a formal description of our approach that builds upon the illustration shown earlier. Our approach verifies each procedure from the specification against the corresponding procedure from the implementation. We represent each process in the specification and the implementation using a *transition diagram* (Definition 5).

We define ϑ to be the set of *visible instructions*. These are the instructions whose semantics we would like preserved between the specification and implementation. Because our algorithm is parameterized by the set ϑ of visible events, we can apply our approach to various settings. For example, in this discussion we consider visible instructions to be input and output to visible channels. In Section 6.8.2, however we define visible instructions to be function calls and return statements. For $v_1, v_2 \in \vartheta$, we write $\langle v_1, \sigma_1 \rangle \equiv \langle v_2, \sigma_2 \rangle$ to represent that v_1 in program state σ_1 is equivalent to v_2 in program states σ_2. In the case of channels, two visible instructions are equivalent iff they both are inputs, or both outputs on the same channel and their values are the same. In the case of function calls and returns, we say that two function calls are equivalent iff the state of globals, the arguments and the address of the called function are the same. Furthermore, we say that two returns are equivalent iff the returned value and the state of the globals are the same. This concept of equivalence for visible instruction can be extended to execution sequences as follows.

Definition 19 (Equivalence of Execution Sequences)**.** *Two execution sequences $\eta_1 \in \mathcal{N}$ and $\eta_2 \in \mathcal{N}$ are said to be equivalent, written $\eta_1 \equiv \eta_2$, if the two sequences contain visible instructions that are pairwise equivalent.*

Definition 20 (Refinement of Transition Diagrams). *Given two transition diagrams* $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ *and* $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$*, we define* π_1 *to be a refinement of* π_2 *(written* $\pi_1 \sqsubseteq \pi_2$*) iff for every* $\sigma_1 \in \Sigma$ *and* $\eta_1\langle \pi_1, \iota_1, \sigma_1\rangle \in \mathcal{N}$ *there exists* $\sigma_2 \in \Sigma$ *and* $\eta_2\langle \pi_2, \iota_2, \sigma_2\rangle \in \mathcal{N}$ *such that* $\eta_1 \equiv \eta_2$*.*

6.5 Simulation Relation

A *verification relation* between two transition diagrams π_1 and π_2 is a set of triples (gl_1, gl_2, ϕ), where $gl_1 \in \mathcal{L}_1$, $gl_2 \in \mathcal{L}_2$ and ϕ is a predicate over the variables live at locations gl_1 and gl_2. Let the set of such predicates be denoted by $\Phi \overset{def}{=} \Sigma \times \Sigma \rightarrow \mathcal{B}$. We write $\phi(\sigma_1, \sigma_2) = true$ to indicate that ϕ is satisfied in $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$.

Simulation relations are verification relations with a few additional properties. To define these properties, we make use of a *cumulative semantic step* relation $\rightsquigarrow^+$, which works like $\rightsquigarrow$, except that it can take multiple steps at once, and it accumulates the steps taken into an execution sequence.

Definition 21 (Cumulative Semantic Step). *Given configurations* $\langle gl_0, \sigma_0\rangle$ *and* $\langle gl_n, \sigma_n\rangle$*, and an execution sequence* η *that contains at least one transition, we define* $\rightsquigarrow^+$ *as follows:*

$$\langle gl_0, \sigma_0\rangle \overset{\eta}{\rightsquigarrow^+} \langle gl_n, \sigma_n\rangle \quad \textit{iff} \quad \eta = \langle gl_0, \sigma_0\rangle \overset{i_1}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n\rangle$$

Definition 22 (Simulation Relation). *A simulation relation* R *for two transition diagrams* $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ *and* $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ *is a verification relation such that:*

$$R(\iota_1, \iota_2, true)$$

$$\begin{array}{l} \forall gl_2, gl_2' \in \mathcal{L}_2, gl_1 \in \mathcal{L}_1, \sigma_1, \sigma_2, \sigma_2' \in \Sigma, \phi \in \Phi, \eta_2 \in \mathcal{N}. \\ \quad \left[\begin{array}{l} \langle gl_2, \sigma_2\rangle \overset{\eta_2}{\rightsquigarrow_2^+} \langle gl_2', \sigma_2'\rangle \wedge \\ \qquad\qquad R(gl_1, gl_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = true \end{array} \right] \Rightarrow \\ \quad \exists gl_1' \in \mathcal{L}_1, \sigma_1' \in \Sigma, \phi' \in \Phi, \eta_1 \in \mathcal{N}. \\ \qquad \left[\begin{array}{l} \langle gl_1, \sigma_1\rangle \overset{\eta_1}{\rightsquigarrow_1^+} \langle gl_1', \sigma_1'\rangle \wedge \\ R(gl_1', gl_2', \phi') \wedge \phi'(\sigma_1', \sigma_2') = true \wedge \eta_1 \equiv \eta_2 \end{array} \right] \end{array}$$

Intuitively, these conditions respectively state that (1) the entry location of π_1 must be related to the entry location of π_2; and (2) if π_1 and π_2 are in a pair of related configurations, and π_2 can proceed one or more steps producing an execution sequence

η_2, then π_1 must also be able to proceed one or more steps, producing a sequence η_1 that is equivalent to η_2, and the two resulting configurations must be related.

The following lemma and theorem connect the above relation with our definition of refinement for transition diagrams (Definition 20).

Lemma 1 (Refinement)**.** *If R is a simulation relation for π_1, π_2, then for each element $(gl_1, gl_2, \psi) \in R$, $\sigma_2 \in \Sigma$, and $\eta_2\langle\pi_2, gl_2, \sigma_2\rangle \in \mathcal{N}$, there exists $\sigma_1 \in \Sigma$, and $\eta_1\langle\pi_1, gl_1, \sigma_1\rangle \in \mathcal{N}$ such that $\eta_1 \equiv \eta_2 \ \wedge \ \psi(\sigma_1, \sigma_2) = true$.*

Theorem 4 (Refinement)**.** *If there exists a simulation relation for π_1, π_2, then $\pi_2 \sqsubseteq \pi_1$.*

The conditions from Definition 22 are used as the base case and the inductive case of a proof by induction showing that π_2 is a refinement of π_1. Thus, a simulation relation is a witness that π_2 is a refinement of π_1.

6.6 Translation Validation Algorithm

Our translation validation algorithms consists of two parts, checking and inference. To show that a transition diagram is a refinement of another transition diagram, we show there exist a simulation relation. In the following sections we describe our algorithms for computing a simulation relation.

Given a transition diagram π and a set of locations S, we define the *skipping transition* relation $\hookrightarrow$, which is a version of $\longrightarrow$ that skips over all locations not in S. This transition allows us to focus our attention on only those locations that are in S.

Definition 23 (Skipping Transition)**.** *Let $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$ be a transition diagram, $gl, gl' \in S$, and $w \in \mathcal{I}^*$, where $w = i_0 \cdots i_n$. We define the skipping transition relation $\hookrightarrow$ for π as follows:*

$$gl \overset{(w,S)}{\hookrightarrow}_{\pi} gl' \quad \textit{iff} \quad \exists gl_1, \cdots, gl_n \in (\mathcal{L} - S). \ \ gl \xrightarrow{i_0} gl_1 \cdots gl_n \xrightarrow{i_n} gl'$$

Throughout the rest of this chapter, we assume that $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ represents the procedure in the specification, and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ represents the corresponding procedure in the implementation. Thus, our goal is to show that π_2 is a refinement of π_1 (i.e. $\pi_2 \sqsubseteq \pi_1$).

6.6.1 Checking Algorithm

In this section, we present the details of our algorithm for checking that a verification relation is indeed a correct simulation relation. We let $\mathscr{R} \subseteq \mathcal{L}_1 \times \mathcal{L}_2 \times \Phi$ to be the verification relation that needs to be checked. We first define two sets of locations $\mathcal{P}_1$ and $\mathcal{P}_2$, which are of interest to our algorithm.

$$\mathcal{P}_1 = \{gl_1 \mid \exists gl_2, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}$$
$$\mathcal{P}_2 = \{gl_2 \mid \exists gl_1, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}$$

To focus our attention on only those locations in $\mathcal{P}_1$ and $\mathcal{P}_2$, we use the skipping transition relation $\hookrightarrow$. In this section, we use the shorthand notation $gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1'$ for $gl_1 \overset{(w_1,\mathcal{P}_1)}{\hookrightarrow}_{\pi_1} gl_1'$, and $gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2'$ for $gl_2 \overset{(w_2,\mathcal{P}_2)}{\hookrightarrow}_{\pi_2} gl_2'$.

Given an entry in $\mathscr{R}$, we then define the *next transition* relation $\twoheadrightarrow$, which traverses the two transition diagrams π_1 and π_2 simultaneously to the next entries reachable from it.

Definition 24 (Next Transition)**.** *Given $(gl_1, gl_2, \phi) \in \mathscr{R}$, $(gl_1', gl_2', \psi) \in \mathscr{R}$, $w_1 \in \mathcal{I}_1^*$ and $w_2 \in \mathcal{I}_2^*$, we define $\twoheadrightarrow$ as follows:*

$$(gl_1, gl_2, \phi) \overset{(w_1,w_2)}{\twoheadrightarrow} (gl_1', gl_2', \psi) \quad \textit{iff} \quad gl_1 \overset{w_1}{\hookrightarrow}_1 gl_1' \wedge gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2'$$

For the verification relation $\mathscr{R}$ to be a simulation relation we require it to satisfy certain conditions. In particular, we want the conditions to make sure that the entry locations are related, and the exit locations are related. Furthermore, the conditions should make sure that for every path in the implementation there is a corresponding path in the specification (our refinement criterion). These conditions are made precise by the following definition of *well-formed relation.* If the relation $\mathscr{R}$ is *not* well-formed, then our checking algorithm immediately rejects the verification relation $\mathscr{R}$.

Definition 25 (Well-Formed Relation)**.** *We define the relation $\mathscr{R}$ to be well formed if the following holds:*

1. $(\iota_1, \iota_2, \mathit{true}) \in \mathscr{R}$
2. $\exists \phi \in \Phi.\ (\epsilon_1, \epsilon_2, \phi) \in \mathscr{R}$
3. $\forall (gl_1, gl_2, \phi) \in \mathscr{R}, gl_2' \in \mathcal{P}_2, w_2 \in \mathcal{I}_2^*$
   $gl_2 \overset{w_2}{\hookrightarrow}_2 gl_2' \implies \exists gl_1' \in \mathcal{P}_1, \psi \in \Phi.\ (gl_1', gl_2', \psi) \in \mathscr{R}$

```
1.  function $\mathsf{CheckRelation}(\mathscr{R})$
2.    for each $(gl_1, gl_2, \phi) \in \mathscr{R}$ do
3.      for each $(gl_1, gl_2, \phi) \xrightarrow{(w_1,w_2)} (gl'_1, gl'_2, \psi)$ do
4.        if $\neg\mathsf{IsInfeasible}(w_1, w_2, \phi)$ then
5.          if $\neg\mathsf{WellPaired}(w_1, w_2, \phi)$ then
6.            Error("Traces are not well formed")
7.          if $\mathsf{ATP}(\phi \Rightarrow \mathsf{wp}(w_1, \mathsf{wp}(w_2, \psi))) \neq \mathit{Valid}$ then
8.            Error("Cannot verify relation entry")

9.  function $\mathsf{IsInfeasible}(w_1 \in \mathcal{I}_1^*, w_2 \in \mathcal{I}_2^*, \phi \in \Phi) : \mathit{Boolean}$
10.   return $\mathsf{ATP}(\neg\mathsf{sp}(w_1, \mathsf{sp}(w_2, \phi))) = \mathit{Valid}$
```

Figure 6.5: Algorithm for checking a simulation relation

The checking algorithm is shown in Figure 6.5. The CheckRelation procedure takes as input a well-formed relation $\mathscr{R}$, and verifies each entry in the verification relation individually. For each possible entry (line 2), the algorithm iterates through all the next transitions as shown in line 3. In doing this search, infeasible paths are pruned out on line 4.

The IsInfeasible function (lines 9-10), checks using an automated theorem prover (ATP) whether or not it is in fact feasible for the specification to follow trace w_1 and the implementation to follow w_2. The trace combination is infeasible if the strongest postconditions (computed using the sp function) with respect to w_2 and then with respect to w_1 is inconsistent. This takes care of pruning within a single program, but also across the specification and the implementation. For a given formula ϕ and trace w, the strongest postcondition $\mathsf{sp}(w, \phi)$ is the strongest formula ψ such that if the instructions in the trace w are executed in sequence starting in a program state satisfying ϕ, then ψ will hold in the resulting program state. The sp computation itself is standard, except for the handling of communication events, which are simulated as assignments. When computing sp with respect to one sequence, we treat all variables from the other sequence as constants. As a result, the order in which we process the two sequences does not matter.

Once we have identified that the two sequences w_1 and w_2 may be a feasible combination, we check that they are well formed using the $\mathsf{WellPaired}$ predicate (lines 5-6). The $\mathsf{WellPaired}$ predicate (not shown here) checks that there is at most one visible instruction in the sequences w_1 and w_2. It also checks that the visible instructions are equivalent.

Next, for well formed sequences, we check that if we start at states that satisfy the predicate ϕ and execute w_1 in π_1 and w_2 in π_2 then the resulting states should satisfy the predicate ψ. To do this we first compute the weakest precondition of ψ with respect to the two traces, and then asks an ATP to show ϕ implies it (line 7). We perform the weakest precondition computation on one trace and then the other. For a given formula ψ and trace w, the weakest precondition $\mathsf{wp}(w, \psi)$ is the weakest formula ϕ such that executing the trace w in a state satisfying ϕ leads to a state satisfying ψ. Here again, the wp computation itself is standard, and the order in which we process the two traces does not matter. If at the end of the algorithm there is no error then the verification relation is indeed a simulation relation.

There are additional optimizations we perform that are not explicitly shown in the algorithm from Figure 6.5. These are however important in improving the efficiency of our refinement checking process. When exploring the control state (both in the checking and in the inference algorithm), we perform a simple partial order reduction [Pel98] that is very effective in reducing the size of the control state space: if two communication events happen in parallel, but they do not depend on each other, and they do not involve externally visible channels, then we only consider one ordering of the two events.

6.6.2 Inference Algorithm

Since there can be many possible paths through a loop, writing simulation relations by hand can be tedious, time consuming and error prone. We therefore need methods for generating these relations automatically, not just checking them. This in turn also allow us to automate the validation process entirely. Nevertheless our checking algorithm is useful by itself, in case our inference algorithm is not capable of finding an appropriate relation, and a human wants to provide the relation by hand.

Here again to focus our attention on only those locations for which our approach infers the relation entries, we define two sets of locations $\mathcal{Q}_1$ and $\mathcal{Q}_2$ for the transition diagrams π_1 and π_2 respectively. These include all locations corresponding to visible events and also all locations before branch statements. In this section, we do notation abuse by re-using the shorthand $gl_1 \xhookrightarrow{w_1}_1 gl'_1$ for $gl_1 \xhookrightarrow{(w_1,\mathcal{Q}_1)}_{\pi_1} gl'_1$, and $gl_2 \xhookrightarrow{w_2}_2 gl'_2$ for $gl_2 \xhookrightarrow{(w_2,\mathcal{Q}_2)}_{\pi_2} gl'_2$.

We now define a parallel transition relation $\hookrightarrow\!\!\!\rightarrow$ that essentially traverses the two transition diagrams (specification and implementation) in synchrony, while focusing on only those locations for which our approach infers the relation entries.

Definition 26 (Parallel Transition). *Given $(gl_1, gl_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$, $(gl'_1, gl'_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$, $w_1 \in \mathcal{I}_1^*$ and $w_2 \in \mathcal{I}_2^*$, we define $\hookrightarrow\!\!\!\rightarrow$ as follows:*

$$(gl_1, gl_2) \overset{(w_1,w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2) \quad \textit{iff}$$
$$gl_1 \xhookrightarrow{w_1}_1 gl'_1 \ \wedge\ gl_2 \xhookrightarrow{w_2)}_2 gl'_2 \ \wedge\ \mathsf{Rel}(w_1, w_2, gl_1, gl_2) \ \wedge\ \mathsf{WellMatched}(w_1, w_2).$$

We now describe the two predicates Rel and WellMatched used in the above definition. The predicate $\mathsf{Rel} : \mathcal{I}^* \times \mathcal{I}^* \times \mathcal{Q}_1 \times \mathcal{Q}_2 \rightarrow \mathcal{B}$ is a heuristic that tries to estimate when a path in the specification is related to a path in the implementation. Consider for example the branch in the specification of Figure 6.1 and the corresponding branch in the implementation. For any two such branches, the Rel function uses heuristics to guess a correlation between them: either they always go in the same direction, or they always go in opposite direction. Using these correlations, $\mathsf{Rel}(w_1, w_2, gl_1, gl_2)$ returns true only if the paths w_1 and w_2 follow branches in a correlated way.

Our implementation of Rel correlates branches in two ways. First, using the results of a strongest postcondition pre-pass over the specification and the implementation, Rel tries to use a theorem prover to prove that certain branches are correlated. If the theorem prover is not able to determine a correlation, Rel uses the structure of the branch predicate and the structure of the instructions on each side of the branch to guess a correlation. For instance, in the example of Figure 6.1, since the strongest postcondition involves the input parameter p, the theorem prover is unable to reason about it. However, because the structure of the branch predicate is not changed in the implementation, Rel can conclude that the two branches go in the same direction.

The other predicate $\mathsf{WellMatched} : \mathcal{I}^* \times \mathcal{I}^* \rightarrow \mathcal{B}$ prunes some of these pair of transitions if the sequence of instructions are not similar (well-matched). We say two sequences (w_1, w_2) of instructions are well-matched if neither of them contain a visible instruction or they each contains a single visible instruction of the same type; i.e. they are both input or both output on the same channel. Although Rel and WellMatched make guesses about the correlation of branches and visible instructions, the later constraint solving phase of our approach makes sure that these guesses are correct.

We now define the relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ of location pairs that will form the entries of our simulation relation.

Definition 27 (Pairs of Interest)**.** *The relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ is defined to be the minimal relation that satisfies the following three properties:*

$$\begin{array}{l} \mathcal{R}(\iota_1, \iota_2) \\ \mathcal{R}(\epsilon_1, \epsilon_2) \\ \mathcal{R}(gl_1, gl_2) \ \wedge \ (gl_1, gl_2) \overset{(w_1,w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2) \quad \Longrightarrow \quad \mathcal{R}(gl'_1, gl'_2) \end{array}$$

The set $\mathcal{R}$ defined above can easily be computed by starting with the empty set, and applying the above three rules exhaustively.

For our approach to successfully infer a simulation relation, the computed set $\mathcal{R}$ must cover every path in the implementation (our refinement criterion). This condition is made precise by the following definition of *well-formed pairs of interest*. The well-formed condition here is similar to the one described in Definition 25, except that now it is for the pairs of interest relation $\mathcal{R}$. We do not need the first two conditions here as they are satisfied by construction. Here again if the computed set $\mathcal{R}$ is *not* well-formed, then our validation approach immediately rejects the translation from specification to implementation.

Definition 28 (Well-Formed Pairs of Interest)**.** *We define the pairs of interest relation $\mathcal{R}$ to be well formed if the following holds:*

$$\begin{array}{l} \forall (gl_1, gl_2) \in \mathcal{R}, gl'_2 \in \mathcal{Q}_2, w_2 \in \mathcal{I}_2^* \\ \qquad gl_2 \overset{w_2}{\hookrightarrow}_2 gl'_2 \implies \exists gl'_1 \in \mathcal{Q}_1.\ (gl'_1, gl'_2) \in \mathcal{R}. \end{array}$$

We now describe our inference algorithm in terms of constraint solving. In particular, for each $(gl_1, gl_2) \in \mathcal{R}$ we define a constraint variable $\psi_{(gl_1,gl_2)}$ representing the predicate that we want to compute for the simulation entry (gl_1, gl_2). We denote by Ψ the set of all such constraint variables. Using these constraint variables, the final simulation relation will have the form:

$$\{(gl_1, gl_2, \psi_{(gl_1,gl_2)}) \mid \mathcal{R}(gl_1, gl_2)\}$$

To compute the predicates that the constraint variables $\psi_{(gl_1,gl_2)}$ stand for, we define a set of constraints on these variables, and then solve the constraints. The constraints are defined as follows.

Definition 29 (Constraint)**.** *A constraint is a formula of the form* $\psi_1 \Rightarrow f(\psi_2)$*, where* $\psi_1, \psi_2 \in \Psi$*, and* f *is a boolean function.*

Definition 30 (Set of Constraints)**.** *The set* $\mathcal{C}$ *of constraints is defined by:*

$$\textit{For each } (gl_1, gl_2) \overset{(w_1,w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2)\text{:}$$
$$\left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{CreateSeed}(w_1, w_2)\right] \in \mathcal{C}$$
$$\left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{wp}(w_1, \mathsf{wp}(w_2, \psi_{(gl'_1,gl'_2)}))\right] \in \mathcal{C}$$

The CreateSeed function above creates for each pair of instruction sequences (w_1, w_2) a formula, which does not refer to any constraint variables. There are two cases either they are well-matched or they are branches (Definition 26). If the instructions are well-matched, then the formula returned by CreateSeed states that the visible instructions in them are equivalent as defined in Section 6.4; and if they are branches, then the formula states the two branches are correlated (either they both go in the same direction, or in opposite directions).

The other function wp used above computes the weakest precondition with respect to w_2 and then with respect to w_1. The weakest precondition computation is the same as the one described in Section 6.6.1.

Having created a set of constraints $\mathcal{C}$, our validation approach now solves these constraints using the algorithm in Figure 6.6. The algorithm starts by setting each constraint variable to *true* (line 13) and initializing a *worklist* with the set of all constraints (line 14). Next, while the *worklist* is not empty, it removes a constraint from the worklist (line 16), and checks using a theorem prover if it is *Valid* (line 17). If not, then it appropriately strengthens the left-hand-side variable of the constraint (line 20) and adds to the worklist all the constraints that have this variable in the right-hand side (lines 21-21).

6.7 Equivalence of Transition Diagrams

Apart from checking refinements, we also sometimes want to check equivalence between two transition diagrams. In this section, we describe how we can generalize

```
11. function SolveConstraints($\mathcal{C}$)
12.   for each $(gl_1, gl_2) \in \mathcal{R}$ do
13.     $\psi_{(gl_1,gl_2)} := true$
14.   let $worklist := \mathcal{C}$
15.   while $worklist$ not empty do
16.     let $[\psi_{(gl_1,gl_2)} \Rightarrow f(\psi_{(gl'_1,gl'_2)})] := worklist.\mathsf{Remove}$
17.     if $\mathsf{ATP}(\psi_{(gl_1,gl_2)} \Rightarrow f(\psi_{(gl'_1,gl'_2)})) \neq Valid$ then
18.       if $(gl_1, gl_2) = (\iota_1, \iota_2)$ then
19.         Error("Start Condition not strong enough")
20.       $\psi_{(gl_1,gl_2)} := \psi_{(gl_1,gl_2)} \wedge f(\psi_{(gl'_1,gl'_2)})$
21.       $worklist := worklist \cup \{c \in \mathcal{C} \mid \exists \psi, g \,.\, c = [\psi \Rightarrow g(\psi_{(gl_1,gl_2)})]\}$
```

Figure 6.6: Algorithm for solving constraints

our algorithms to check for equivalence. We first define two transition diagrams to be equivalent as follows:

Definition 31 (Equivalence of Transition Diagrams)**.** *Two transition diagrams π_1 and π_2 are said to be equivalent iff $\pi_1 \sqsubseteq \pi_2$ and $\pi_2 \sqsubseteq \pi_1$.*

We define a *bisimulation relation* using the definition of simulation relation.

Definition 32 (Bisimulation Relation)**.** *A verification relation R is a bisimulation relation for π_1, π_2 iff R is a simulation relation for π_1, π_2 and $R^{-1} = \{(gl_2, gl_1, \phi) \mid R(gl_1, gl_2, \phi)\}$ is a simulation relation for π_2, π_1.*

The following theorem connects the above relation with our definition of equivalence for transition diagrams.

Theorem 5 (Equivalence)**.** *If there exists a bisimulation relation for π_1, π_2, then π_1 and π_2 are equivalent.*

Like simulation relation, a bisimulation relation is a witness that two transition diagrams are equivalent. Therefore, to check if the specification is equivalent to the implementation our algorithms now have to show that there exists a bisimulation relation between them. We can use both our checking and inference algorithms for this purpose with just slight modifications.

For the checking algorithm, we only have to strengthen the definition of well-formed relation (Definition 25) with this fourth condition.

$$\begin{aligned} &\forall (gl_1, gl_2, \phi) \in \mathscr{R}, gl'_1 \in \mathcal{P}_1, w_1 \in \mathcal{I}_1^* \\ &\quad gl_1 \xhookrightarrow{w_1}_1 gl'_1 \implies \exists gl'_2 \in \mathcal{P}_2, \psi \in \Phi.\ (gl'_1, gl'_2, \psi) \in \mathscr{R}. \end{aligned}$$

Similarly, for the inference algorithm, we only have to strengthen the definition of well-formed pairs of interest (Definition 28) with this condition.

$$\begin{aligned} &\forall (gl_1, gl_2) \in \mathcal{R}, gl'_1 \in \mathcal{Q}_1, w_1 \in \mathcal{I}_1^* \\ &\quad gl_1 \xhookrightarrow{w_1}_1 gl'_1 \implies \exists gl'_2 \in \mathcal{Q}_2.\ (gl'_1, gl'_2) \in \mathcal{R}. \end{aligned}$$

6.8 Experiments and Results

We implemented our algorithms in a tool called Surya using the Simplify ATP [DNS05]. We have used Surya to validate programs in two different settings. First, we used it to *automatically* check refinements of various concurrent programs, written in CSP. Next, we used Surya to validate the result of the high-level synthesis framework Spark.

6.8.1 Automatic Refinement Checking of CSP Programs

For refinements our goal is to infer a simulation relation (if possible). The visible events in this case are input and output on visible channels. We wrote a variety of CSP refinements, and checked them for correctness automatically. The refinements that we checked are shown in Table 6.3, along with the number of parallel threads, the number of instructions, the number of simulation relation entries, the number of calls to the theorem prover, and the time required to automatically check them. Apart from the theorem prover calls discussed in this chapter, we also use the theorem prover to reduce the size of the formulas used in our algorithms. The number of calls to the theorem prover mentioned in Table 6.3 include *all* these calls.

The first 11 refinements were inspired from examples that come with the FDR tool [Ltd]. FDR is a state-of-the-art tool to check CSP refinements. The approach that FDR uses for checking refinement is to perform an exhaustive search of the

Table 6.3: Timings for the refinement examples checked using our tool

Description	T	I	SRE	TP	Time mins
1. Simple buffer	7	29	3	14	00.00
2. Simple vending machine	2	20	9	32	00.00
3. Cyclic scheduler	6	65	157	11082	00.49
4. Student tracking system	3	63	12	115	00.01
5. 1 comm link	11	54	3	14	00.01
6. 2 parallel comm links	18	105	37	486	00.04
7. 3 parallel comm links	25	144	45	1861	00.21
8. 4 parallel comm links	32	186	124	7228	01.11
9. 5 parallel comm links	39	228	315	24348	02.32
10. 6 parallel comm links	46	270	762	74991	08.29
11. 7 parallel comm links	53	312	1785	217131	37.28
12. SystemC refinement	8	39	3	14	00.00
13. EP2 system	3	173	208	5648	01.47
T: Number of parallel Threads.					
I: Number of Instructions.					
SRE: Number of Simulation Relation Entries.					
TP: Number of Theorem Prover calls.					

implementation-specification combined state space. Although in its pure form this approach only works for finite state systems, there is one way in which it can be extended to infinite systems. In particular, if an infinite state system treats all the data it manipulates as black boxes, then one can use skolemization and simply check the refinement for one possible value. Such systems are called *data-independent*, and FDR can check the refinement of these systems using the skolemization trick, even if they are infinite [RV01].

Unfortunately, for high-level programs, there are many refinement examples that are not finite, because they do not specify the bit-width of integers (in particular, we want the refinement to work for any integer size). Nor are the processes data-independent, as they manipulate the data during the refinement process. In particular, our example from Figure 6.1 is neither finite nor data-independent, since both the specification and the implementation are "inspecting" the variables when manipulating them. Indeed, it would not at all be safe to simply check the refinement for any one particular value, since, if we happen to pick 0 for p, and the implementation erroneously sets the output to 4 times the input (instead of 2 times), we would not detect the error. FDR cannot check the refinement of such infinite data-dependent CSP systems, except by restricting them to a finite subset first, for example by picking a bit-width for the integers, and then doing an exhaustive search. Not only would such an approach not prove the refinement for any bit-width, but furthermore, despite many techniques that have been developed for checking larger and larger finite state spaces [BCM$^+$90, CD93, Pel98, RGG$^+$95], the state space can still grow to a point where automation is impossible. For example, we tried checking the refinement example '2 parallel comm links' from Table 6.3 in FDR using 32-bit integers as values, and the tool had to be stopped because it ran out of memory after several hours (Our tool, in contrast, is able to check this example for any sized integers, not just 32-bit integers, in about 4 seconds).

We implemented generalizations of these 11 FDR examples to make them data-dependent and operate over infinite domains. We were able to check these generalized refinements that FDR would not be able to check.

The 12th refinement in the list is a hardware refinement example taken from a SystemC book [GLMS02]. This example models the refinement of an abstract FIFO communication channel to an implementation that uses a standard FIFO hardware channel, along with logic to make the hardware channel correctly implement the abstract communication channel.

In the 13th refinement from Table 6.3, we checked part of the EP2 system [EP], which is a new industrial standard for electronic payments. We followed the implementation of the data part of the EP2 system found in a recent TACAS 2005 paper on CSP-Prover [IR05]. The EP2 system states how various components, including service centers, credit card holders, and terminals, interact.

In all of the above examples, we check for trace subset refinement (see Definition 20). Since trace subset refinement preserves safety properties, we can also conclude

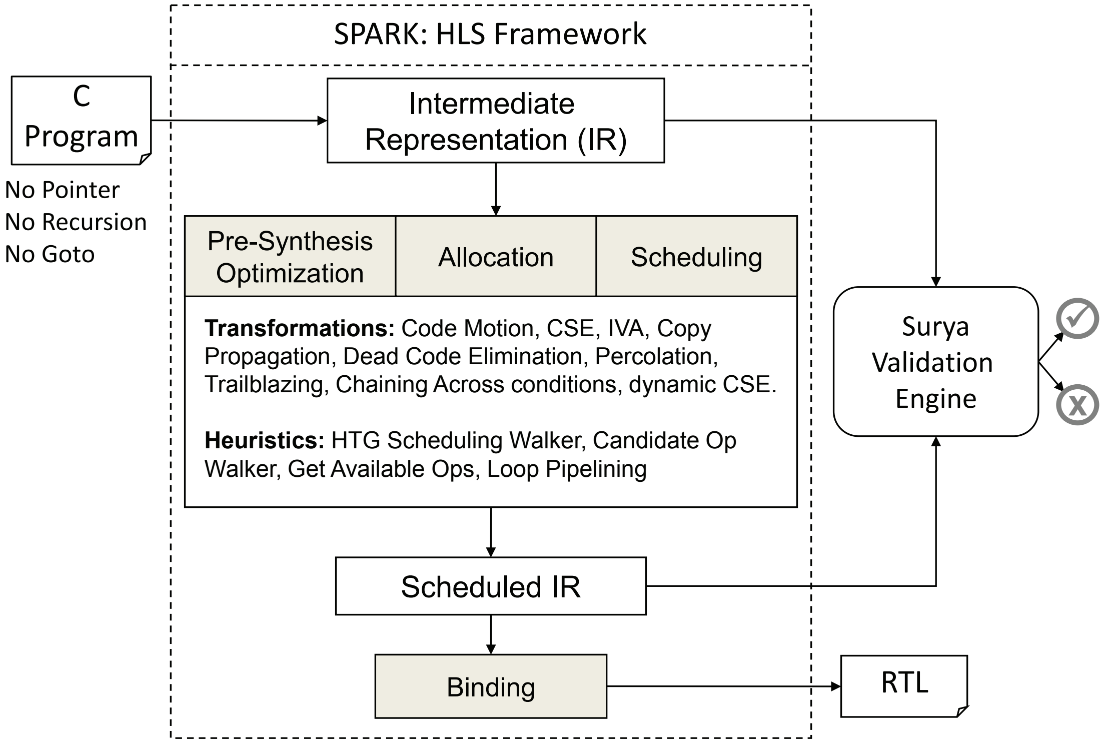

SPARK: HLS Framework
C Program
No Pointer
No Recursion
No Goto
Intermediate Representation (IR)
Pre-Synthesis Optimization
Allocation
Scheduling
Transformations: Code Motion, CSE, IVA, Copy Propagation, Dead Code Elimination, Percolation, Trailblazing, Chaining Across conditions, dynamic CSE.
Heuristics: HTG Scheduling Walker, Candidate Op Walker, Get Available Ops, Loop Pipelining
Scheduled IR
Binding
RTL
Surya Validation Engine

Figure 6.7: Overview of the Spark framework along with Surya

that the implementation has all the safety properties of the specification.

We also have a large test suite of incorrect refinements that we run our tool on, to make sure that our tool indeed detects these as incorrect refinements.

6.8.2 SPARK: High-Level Synthesis Framework

Spark is a C-to-VHDL parallelizing high-level synthesis framework that employs a set of compiler, parallelizing compiler, and synthesis transformations to improve the quality of high-level synthesis results. Figure 6.7 shows an overview of the Spark HLS framework. What makes Spark an excellent candidate for experimenting is not only the easy availability of source code but also the fact that it uses a single intermediate representation, called Hierarchical Task Graphs (HTGs) [GP92]. Spark starts with a behavioral description in ANSI-C as input – currently with the restrictions of no pointers, no recursion, and no irregular control-flow jumps. It converts the input program into its own IR, and then applies a set of code transformations, including loop unrolling,

Table 6.4: Spark benchmarks successfully checked

Benchmarks	No. of bisimulation relation entries	No. of calls to theorem prover	Time secs
1. Incrementer	6	9	00.52
2. Integer-sum	6	20	00.81
3. Array-sum	6	24	00.83
4. Diffeq	7	41	01.68
5. Waka	11	79	02.61
6. Pipelining	12	75	02.30
7. Rotor	14	71	02.57
8. Parker	26	281	05.23
9. S2r	27	570	26.73
10. Findmin8	29	787	14.86

loop fusion, common sub-expression elimination, copy propagation, dead code elimination, loop-invariant code motion, induction variable analysis, and operation strength reduction. Following these transformations, Spark performs a scheduling phase using resource allocation information provided by the user. This scheduling phase also performs a variety of transformations, including speculative code motion, dynamic renaming of variables, dynamic branch balancing, chaining of operations across conditional blocks, and scheduling on multi-cycle operations. The scheduling phase is followed by a resource binding phase and finally by a back-end code generation pass that produces RTL VHDL.

Our tool Surya takes as input the IR program that is produced by the parser, and the IR program right before resource binding (see Figure 6.7), and verifies that the two are equivalent by showing that there exist a bisimulation relation. Our tool therefore validates the entire HLS process of Spark, except for parsing, resource binding and code generation. Note that Surya is around 7,500 lines of C++ code, whereas Spark's implementation excluding the parser consists of over 125,000 lines of C++ code. Thus, with around 15 times less effort compared to Spark's implementation we can build a tool that validates its synthesis process.

We tested our tool on 12 benchmarks obtained from Spark's test suite. Of these benchmarks, 10 passed and 2 failed. The benchmarks that were succesfully checked are shown in Table 6.4, along with the number of bisimulation relation entries, the number of calls to the theorem prover, and the time required to check each benchmark. All these benchmarks are single threaded. For the ones that passed, our tool was able to quickly find the bisimulation relation, taking on average around 6 seconds per procedure, and a maximum of 27 seconds for the largest procedure (80 lines of code). Furthermore, the computed bisimulation relations were small, ranging in size from 6 to 29 entries, with an average of about 14. To infer these bisimulation relations, our approach made an average of 189 calls to the theorem prover per procedure (with a minimum of 9 and a maximum of 797). Our approach is compositional since it works on one procedure at a time, and the above results show that our approach can handle realistically size procedures.

As mentioned previously, two benchmarks failed our validation test. Upon further analysis each of them lead us to discover previously unknown bugs in Spark. One bug occurs in a particular corner case of copy propagation for array elements. The other bug is in the implementation of the code motion algorithm in the scheduler. The fact that Surya found two previously unknown bugs in a widely-used HLS framework emphasizes the usefulness and bug-isolating capabilities of our tool.

In general, our tool will perform well when the transformations that are performed preserve most of the program's control flow structure. Such transformations are called *structure-preserving transformations* [ZPFG03]. The only non structure-preserving transformation that Spark performs is loop unrolling, but in our examples this transformation did not trigger.

6.9 Related Work

Our work is related to translation validation [PSS98, Nec00, RM99, ZPFG03, GZB05, ZPG$^+$05, KLG07], relational approaches to reasoning about programs [FV99, BG00, LJWF02, Ben04, Jos88], CSP refinement checking [Ltd, DS97, TW97, IR05], and HLS verification [ABRM98, EHR99, NTR$^+$01, KKM04, KLG08]. We now discuss each area in more detail.

Translation Validation: Our inference algorithm was inspired by Necula's translation validation algorithm for inferring simulation relations that prove equivalence of sequential programs [Nec00]. Necula's approach collects a set of constraints in a forward scan of the two programs, and then solves these constraints using a specialized solver and expression simplifier. Unlike Necula's approach, our algorithm must take into account statements running parallel, since hardware is inherently concurrent and one of the main tasks that HLS tools perform is to schedule statements for parallel execution. Furthermore our algorithm is expressed in terms of calls to a general theorem prover, rather than using specialized solvers and simplifiers. In this sense our algorithm is more modular, since the theorem proving part of the algorithm has been modularized into a component with a very simple interface (it takes a formula and returns *Valid* or *Invalid*). This allows us to easily substitute the current Simplify theorem prover with another one.

Relational approaches: Relational approaches are a common tool for reasoning about programs, and they have been used for a variety of verification tasks, including model checking [FV99, BG00], translation validation [PSS98, Nec00], and reasoning about optimizations once and for all [LJWF02, Ben04]. In this context, our work is inspired by Josephs's approach [Jos88] for proving refinements. However, Josephs proved refinements by hand, whereas our tool is fully automated.

CSP refinement checking: There has been a long line of work on reasoning about refinement of CSP programs. Our searching algorithm through the control state of the program is similar to FDR's searching technique [Ltd], which exhaustively explores the state space. However, as mentioned previously, our tool can handle infinite state spaces that do not trivially reduce using skolemization to finite state spaces. We achieve this by capturing the possibly infinite state space of data using formulas and using a theorem prover to reason about these formulas. Although, this technique is well known and has been used in dataflow analysis [GS97, FLL$^+$02], model checking [CCG03, HJMS02, BMMR01], and translation validation [PSS98, Nec00]. The use of this technique in the context of checking CSP refinements appears to be novel.

Various interactive theorem provers have been extended with the ability to reason about CSP programs. As one example, Dutertre and Schneider [DS97] reasoned about communication protocols expressed as CSP programs using the PVS theorem prover [ORR$^+$96]. As another example, Tej and Wolff [TW97] have used the Isabelle theorem prover [Pau94] to encode the semantics of CSP programs. Isabelle has also been used by Isobe and Roggenbach to develop a tool called CSP-PROVER [IR05] for proving properties of CSP programs. All these uses of interactive theorem provers follow a common high-level approach: the semantics of CSP is usually encoded using the native logic of the interactive theorem prover, and then a set of tactics are defined for reasoning about this semantics. Users of the system can then write proof scripts that use these tactics, along with built-in tactics from the theorem prover, to prove properties about particular CSP programs. Our approach does not have the same level of formal underpinnings as these interactive theorem proving approaches. However, our approach is fully automated, whereas these interactive theorem proving approaches all require some amount of human intervention.

Our tool checks one particular property of CSP programs, namely trace subset refinement. This kind of refinement only preserves safety properties. Algorithms and tools exist for checking other kinds of refinements. For example, CSP-Prover [IR05] can check refinements using a failures semantics that preserves liveness properties and deadlock freedom (in addition to safety properties). The FDR [Ltd] tool can also check refinements in a failures/divergence model, which can also preserve livelock freedom.

HLS verification: Techniques like correctness-preserving transformations [EHR99], formal assertions [NTR$^+$01], symbolic simulation [ARGB99], and relational approaches for functional equivalence of FSMDs [KKM04, KMS$^+$06] have been used to validate the scheduling step of HLS. However, all these techniques assume that the scheduler does not move code across basic blocks and variable names do not change, which would prevent them from validating Spark's HLS process. Also, in work that is complementary to ours, model checking was used to validate the binding step of HLS [ABRM98], which is the only internal step of Spark that our tool does not validate.

6.10 Summary

In this chapter, we have presented an automated algorithm for translation validation of the HLS process. We have implemented our algorithm in a validation system called Surya and demonstrated its effectiveness through its application in two different settings. The innovation in our work lies in showing that translation validation approaches work well in the application domain of high-level synthesis.

Our experiments with Spark showed that with only a fraction of the development cost of Spark, our algorithm can validate the translations performed by Spark, and it also uncovered bugs that eluded long-term use. Our work also solves the critical problem of handling more sophisticated datatypes than finite bit-width enumeration types associated with typical RTL code and thus enables stepwise refinement of system designs expressed using high-level languages. In the next chapter we discuss how we can generalize this translation validation algorithm to perform once-and-for-all validation of parts of HLS tools.

Acknowledgements: Chapter 6, in part, has been published as “Automated Refinement Checking of Concurrent Systems” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *ICCAD 07: Proceedings of the 2007 IEEE/ACM International Conference on Computer Aided Design* [KLG07]. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, has been published as “Validating High-Level Synthesis” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta in *CAV 08: Proceedings of the 20th international conference on Computer Aided Verification* [KLG08]. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, is in preparation for publication as “Translation Validation of High-Level Synthesis” by Sudipta Kundu, Sorin Lerner and Rajesh Gupta. The dissertation author is the primary investigator and author of this paper.

Chapter 7

Parameterized Program Equivalence Checking

In the previous chapter we discussed an approach to verify if two programs are equivalent, thereby proving that the translation (performed by an HLS tool) from high-level design to low-level design is correct. In this chapter, we discuss another approach that guarantees correctness of the translation from high-level design to low-level design, by proving the HLS tool itself correct. Unlike translation validation, this approach proves the correctness of an HLS tool *once and for all*, before it is ever run. In the following sections we describe in details our approach called *Parametrized Equivalence Checking* (PEC) that generalizes the translation validation approach discussed in the previous chapter to automatically establish the correctness of semantics preserving transformations once and for all.

7.1 Overview of Synthesis Tool Verification

HLS tools are a fundamental component of the tool chains hardware designers rely on for system-level designs. As a result, correctness of HLS tool is crucially important. A bug in a HLS tool can in turn introduce errors in each generated RTL. Furthermore, HLS tool bugs can invalidate strong guarantees that were established on the original source program (discussed in Chapter 4 and 5). Unfortunately, as discussed throughout this thesis building reliable compilers is difficult, error-prone, and requires significant manual effort.

One of the most error prone parts of a HLS tool is its optimization phase. Many optimizations require an intricate sequence of complex transformations. Often these transformations interact in unexpected ways, leading to a combinatorial explosion in the number of cases that must be considered to ensure that the optimization phase is correct.

Once-And-For-All vs. Translation Validation

Previous techniques for providing correctness guarantees for optimizations can be divided into two categories: *once and for all* and *translation validation*.

The primary advantage of once-and-for-all techniques is that they provide a very strong guarantee: optimizations are known to be correct when the tool is built, before they are run even once. In contrast, translation validation provides a weaker correctness guarantee. This is because translation validation guarantees that only a particular run of the optimization is correct. Tools that include translation validation may still contain bugs and it is unclear what a designer should do when the translation validator flags a particular translation to be incorrect.

On the other hand, translation validation techniques have a clear advantage over once-and-for-all techniques in terms of automation. Most of the techniques that provide once-and-for-all guarantees require user interaction. Those that are fully automated, for example Cobalt [LMC03] and Rhodium [LMRC05] approaches for compilers, work by having programmers implement optimizations in a domain-specific language using flow functions and single-statement rewrite rules. Unfortunately, the set of optimizations that these techniques can prove correct has lagged behind translation validation. In particular, translation validation can already handle complex loop optimizations like skewing, splitting and interchange, which have thus far eluded automated once-and-for-all approaches. A common intuition is that once-and-for-all proofs are harder to achieve because they must show that any application of the optimization is correct, as opposed to a single instance.

7.2 Overview of Our Approach

In this chapter, we present a new technique for proving optimizations correct called Parameterized Equivalence Checking (PEC) that bridges the gap between translation validation and once-and-for-all techniques. PEC generalizes translation validation to handle parameterized programs, which are partially specified programs that can represent multiple concrete programs. For example, a parameterized program may contain a section of code whose only known property is that it does not define or use a particular variable.

The key insight of PEC is that existing translation validation techniques can be adapted to work in the broader setting of parameterized programs. This allows translation validation techniques, which have traditionally been used to prove *concrete* programs equivalent, to prove *parameterized* programs equivalent. Most importantly, because optimizations can be expressed as nothing more than parameterized transformation rules, using before and after parameterized code patterns, PEC can prove once and for all that such optimizations preserve semantics.

To highlight the power and generality of PEC, we designed a new language for writing optimizations, and implemented a checker based on PEC that can automatically check the correctness of optimizations written in this language. Our language for implementing and proving optimizations correct is much more expressive than previous such optimization languages, like Cobalt [LMC03] and Rhodium [LMRC05]: whereas Cobalt and Rhodium only supported local rewrites of a single statement to another, our language supports many-to-many rewrite rules. Such rules are able to replace an entire set of statements, even entire loops and branches, with a completely different set of statements. Using these rules, we can express many more optimizations than in Cobalt and Rhodium, and we can also prove them all correct using our PEC algorithm.

Contributions

The main contributions of our approach are:

- We developed a technique for performing Parameterized Equivalence Checking. PEC adapts the traditional translation validation approach presented in Chapter 6 to the setting of once-and-for-all correctness proofs.

- We developed a new language for implementing optimizations. Our language is more expressive than previous languages that can be checked for correctness automatically: it has explicit support for expressing many-to-many transformations, meaning that a set of statements can be transformed to another set of statements in a single rewrite.

```
i := 0
while (i < n) {
   a[i] += 1;
   b[i] += a[i];
   i++;
}
```

(a)

```
i := 0
a[i] += 1;
while (i < n - 1) {
   b[i] += a[i];
   i++;
   a[i] += 1;
}
b[i] += a[i];
i++;
```

(b)

Figure 7.1: Loop pipelining: (a) original code, and (b) optimized code

- We implemented and proved correct a variety of complex optimizations in our system, which have been difficult or impossible to prove in previous systems.

7.3 Illustrative Example

In the previous chapter we discussed the validation of the Spark [GDGN03] HLS tool. In Spark it was shown that compiler optimizations and various other source-to-source transformations can be extremely useful for generating highly parallel designs. In this section we illustrate the main ideas of our approach through one such compiler optimizations: loop pipelining. Loop pipelining can break dependencies inside a loop body without increasing the code size of the loop body, and thus provides more flexibility during the scheduling phase of HLS.

As an example, consider the code in Figure 7.1(a). The statements in the original loop cannot be scheduled together (to be executed in parallel) as there is a dependency between them – the instruction `b[i] += a[i]` must wait until the instruction `a[i] += 1` finishes. Figure 7.1(b) shows the result of applying loop pipelining on this loop. The statements in this transformed loop does not depend on each other and can be scheduled together. In particular, the instruction `b[i] += a[i]` can be scheduled with the instructions `i++; a[i] += 1`. However, to make this transformation correct, one has to add a prologue at the beginning of the transformed loop in order to setup the pipelining

$$\left[\begin{array}{ll} & \mathbf{I}\ \texttt{:= 0} \\ L_1: & \texttt{while}\ (\mathbf{I} \texttt{<} \mathbf{E})\ \texttt{\{} \\ L_2: & \qquad \mathbf{S_1} \\ L_3: & \qquad \mathbf{S_2} \\ L_4: & \qquad \mathbf{I}\texttt{++} \\ & \texttt{\}} \end{array}\right] \Longrightarrow \left[\begin{array}{l} \mathbf{I}\ \texttt{:= 0} \\ \mathbf{S_1} \\ \texttt{while}\ (\mathbf{I} \texttt{<} \mathbf{E}\texttt{-1})\ \texttt{\{} \\ \qquad \mathbf{S_2} \\ \qquad \mathbf{I}\texttt{++} \\ \qquad \mathbf{S_1} \\ \texttt{\}} \\ \mathbf{S_2} \\ \mathbf{I}\texttt{++} \end{array}\right]$$

where

$$\begin{array}{l} \mathit{DoesNotModify}(\mathbf{S_1}, \mathbf{I})@L_2 \ \wedge\ \mathit{DoesNotModify}(\mathbf{S_2}, \mathbf{I})@L_3 \ \wedge \\ \mathit{StrictlyPositive}(\mathbf{E})@L_1 \ \wedge\ \mathit{DoesNotModify}(\mathbf{S_1}, \mathbf{E})@L_2 \ \wedge \\ \mathit{DoesNotModify}(\mathbf{S_2}, \mathbf{E})@L_3 \ \wedge\ \mathit{DoesNotModify}(\mathbf{I}\texttt{++}, \mathbf{E})@L_4 \end{array}$$

Figure 7.2: Loop pipelining transformation

fact *StrictlyPositive*($\mathbf{E}$)
has meaning $\mathit{eval}(\sigma, \mathbf{E}) > 0$

fact *DoesNotModify*($\mathbf{S}, \mathbf{E}$)
has meaning $\mathit{eval}(\sigma, \mathbf{E}) = \mathit{eval}(\mathit{step}(\sigma, \mathbf{S}), \mathbf{E})$

Figure 7.3: Meanings of some facts that we use in our system

effect. There is also an epilogue after the loop to execute the remaining instructions.

7.3.1 Expressing Loop Pipelining

We implement loop pipelining in our language as shown in Figure 7.2. The transformation simply moves some instructions (namely $\mathbf{S_1}$) from the current iteration to the next iteration. Optimizations in our language are written as parameterized rewrite rules with side conditions: $P_1 \Longrightarrow P_2$ **where** ϕ, where P_1 and P_2 are parameterized programs, and ϕ is a side condition that states when the rewrite rule can safely be fired. An optimization $P_1 \Longrightarrow P_2$ **where** ϕ states that when a concrete program is found that matches the parameterized program P_1, it should be transformed to P_2 if the side condition ϕ holds.

Parameterized Programs

A parameterized program is a partially specified program that can represent multiple concrete programs. For example, in the original and transformed programs from Figure 7.2, $\mathbf{S_1}$ ranges over concrete statements (including branches, loops, and sequences of statements) that are single-entry-single exit; **I** ranges over concrete program variables; and **E** ranges over concrete expressions. Because variables like $\mathbf{S_1}$, **I** and **E** range over the syntax of concrete programs, we call such variables *meta-variables*. To simplify exposition, rather than provide explicit types for all meta-variables, we instead use the following naming conventions: meta-variables starting with **S** range over statements, meta-variables starting with **E** range over expressions, and meta-variables starting with **I** range over variables.

Side Conditions

The side conditions are boolean combinations of facts that must hold at certain points in the original program. For example the side condition $\mathit{DoesNotModify}(\mathbf{S_1}, \mathbf{I})@L_2$ in Figure 7.2 states that at location L_2 in the original program $\mathbf{S_1}$ should not modify **I**. In general, side conditions are first-order logic formulas with facts like $\mathit{DoesNotModify}(\mathbf{S_1}, \mathbf{I})@L_1$ as atomic predicates.

Each fact used in the side condition must have a semantic meaning, which is a predicate over program states. Figure 7.3 gives the semantic meanings for the two primary facts that we use in our system. In general, meanings can be first-order logic formulas with a few special function symbols: (1) σ is a term that represents the program state at the point where the fact holds. (2) *eval* evaluates an expression in a program state and returns its value; (3) *step* executes a statement in a program state and returns the resulting program state.

The semantic meanings are used by the PEC algorithm to determine the semantic information that can be inferred from the side conditions when proving correctness. Although optimization writers must provide these meanings, in our experience we have found that there is a small number of common facts used across many different optimizations (for example *DoesNotModify*), and since these meanings only need to be written once, the effort in writing meanings is not onerous.

Executing Optimizations

Optimizations written in our language are meant to be executed by an execution engine. When running an optimization $P_1 \Longrightarrow P_2$ **where** ϕ, the execution engine must find concrete program fragments that match P_1. Furthermore, it must perform some program analysis to determine if the facts in the side condition ϕ hold. One option for implementing these program analysis is to use a general purpose programming language. Although this provides the most flexibility, it does not guarantee that the facts in the side condition are computed correctly. Alternatively, if one wants stronger correctness guarantees, the facts in the side conditions can be computed in a way that guarantees that their semantic meanings hold, for example using the Rhodium system of Lerner *et al.* [LMRC05], or using Leroy's Compcert system [Ler06]. Note that it is straight forward to see how the rewrite rule in Figure 7.2 performs loop pipelining on our example in Figure 7.1.

7.3.2 Proving Correctness of Loop Pipelining

Our goal is to show that the loop pipelining optimization written in our language is correct, once and for all, before it is even run once. To do this, we must show that the rewrite rule from Figures 7.2 satisfies the following property: given the side conditions, the original parameterized program and the transformed parameterized program have the same behavior. We next discuss our approach in details.

Parameterized Equivalence Checking

In Chapter 6 we discussed Translation Validation (TV), a technique that proves concrete, fully specified programs equivalent. In our setting, we are attempting to prove parameterized programs equivalent. To achieve this, we developed a technique called Parameterized Equivalence Checking (PEC) that generalizes traditional TV techniques to the setting of parameterized programs.

There are two simple observations that intuitively explain why techniques from translation validation can be generalized to parameterized programs. The first observation is that if a program fragment **S** in the original program executes in a program state σ, and the same program fragment **S** executes in the transformed program in the same state σ, then we know that the two resulting states are equal. This shows that we can reason about state equality even if we don't know what the program fragments are. The second observation is that when proving equivalence, we are usually interested in some key invariants that justify the optimization. The insight is that the semantic meaning of the side condition captures precisely when these key invariants can be propagated throughout statements that are not fully specified. For example, if the correctness of an optimization really depends on **I** not being modified in a region of code, the side condition will allow us to know this fact, and thus reason about **I** across such unknown statements.

Bisimulation Relation

PEC proves equivalence using *bisimulation relation* (Definition 32). Recall that bisimulation relation is defined in terms of the more basic concept of *verification relations*, which is a set of entries of the form (gl_1, gl_2, ϕ), where each entry relates a program point gl_1 in the original program with a corresponding program point gl_2 in the transformed program, and the predicate ϕ indicates how the state of the two programs are related at that point. Moreover, a bisimulation relation is simply a verification relation that satisfies the property that the predicate on any entry in the relation implies the predicate on all entries reachable from it. In the following we denote the original parameterized program as the specification and the transformed parameterized program as the implementation.

The PEC approach generalize the inference algorithm described in the previous chapter. Similar to the TV algorithm, our PEC approach works in two steps. In the first step we start by finding pairs of locations in the specification and the implementation that need to be related in the bisimulation and then for each pair of locations (gl_1, gl_2), we define a constraint variable $\psi(gl_1, gl_2)$ to represent the state-relating formula that will be computed in the bisimulation relation for that pair. We next define a set of constraints over these variables that must be satisfied in order for the would-be bisimulation relation to in fact be a bisimulation. In the second step, we solve the constraints using an iterative algorithm.

Figure 7.4 shows the concurrent control flow graph (CCFG) for the two parameterized programs of our example, along with the related locations that our approach generates. The related locations are shown using a dashed line (labelled A – F), and each entry has a predicate associated with it which is shown in Table 7.1. These predi-

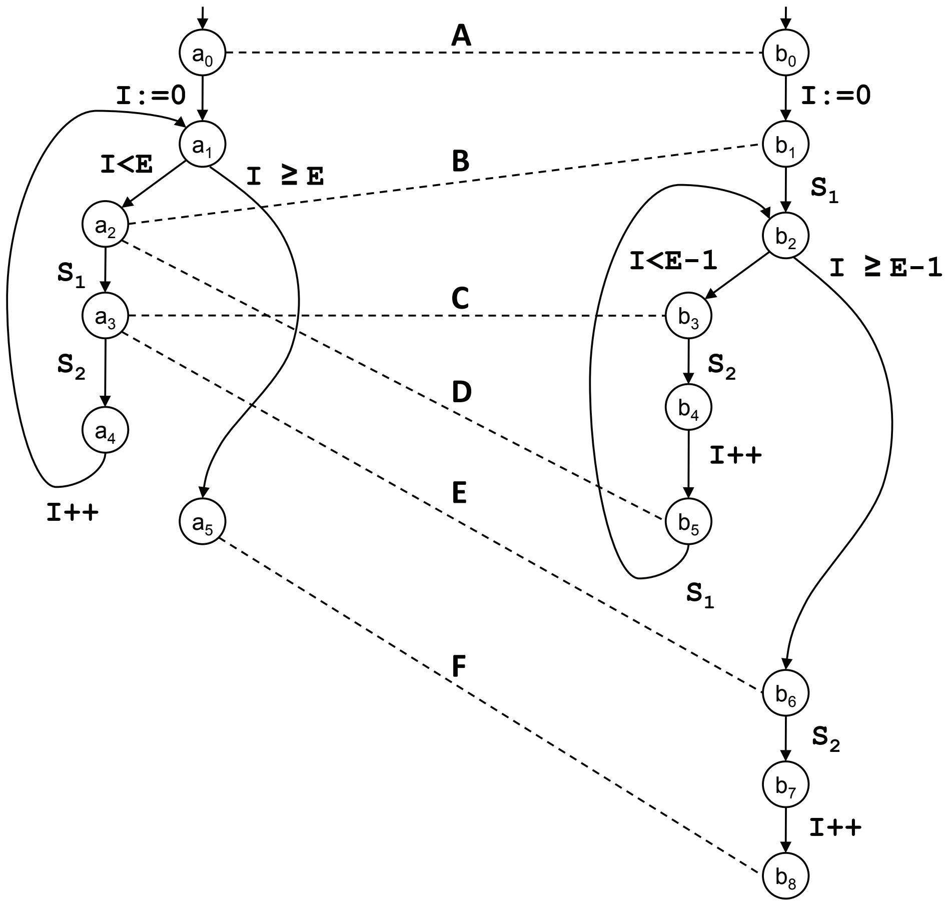


Figure 7.4: CCFGs of running example with the related locations

cates operate over the program states σ_1 and σ_2 of the specification and implementation programs. To make the notation cleaner, we use some shorthand notation. For example, $\mathbf{E}_1$ means $eval(\sigma_1, \mathbf{E})$. Using this notation, the predicate at C states that (1) the two programs states σ_1 and σ_2 are equal, (2) $\mathbf{I} < \mathbf{E}$ holds in σ_1 and (3) $\mathbf{I} < \mathbf{E} - 1$ holds in σ_2. Together the related locations and the predicates form the bisimulation relation for this example.

Generating Constraints

Our PEC algorithm first relate the start locations of the two programs and then adds the constraint that the constraint variable at A ($\psi_{(a_0,b_0)}$) should imply the predicate $\sigma_1 = \sigma_2$; this indicates that we can assume the program states are equal at the start

Table 7.1: A bisimulation relation for our running example

$(gl_{\mathbf{1}}, gl_{\mathbf{2}})$	ϕ
A. (a_0, b_0)	$\sigma_1 = \sigma_2$
B. (a_2, b_1)	$\sigma_1 = \sigma_2 \ \wedge \ \mathbf{I_1} < \mathbf{E_1}$
C. (a_3, b_3)	$\sigma_1 = \sigma_2 \ \wedge \ \mathbf{I_1} < \mathbf{E_1} \ \wedge \ \mathbf{I_2} < \mathbf{E_2} - 1$
D. (a_2, b_5)	$\sigma_1 = \sigma_2 \ \wedge \ \mathbf{I_1} < \mathbf{E_1} \ \wedge \ \mathbf{I_2} < \mathbf{E_2}$
E. (a_3, b_6)	$\sigma_1 = \sigma_2 \ \wedge \ \mathbf{I_1} < \mathbf{E_1} \ \wedge \ \mathbf{I_2} \geq \mathbf{E_2} - 1$
F. (a_5, b_8)	$\sigma_1 = \sigma_2$

of the two code fragments. Similarly, it also adds the constraint $\psi_{(a_5,b_8)} \Rightarrow (\sigma_1 = \sigma_2)$ corresponding to the end locations (F); this constraint indicates that we must establish that the program states are equal after the two programs execute. To generate the locations in between, our algorithm traverses both programs in parallel from the top entry. Each time a statement is reached like $\mathbf{S_1}$, and $\mathbf{S_2}$, the algorithm finds the corresponding location in the other program, and adds a relation entry between the two locations with the corresponding constraint variable implying the predicate $\sigma_1 = \sigma_2$ (since this is the only mechanism we have to preserve equivalence of arbitrary statements). Apart from these, our algorithm also generates constraints such that the constraint variable for each related pair that is under a branch, implies the strongest post condition of the branch condition. These constraints lead to the various predicates relating **I** and **E** in Table 7.1. This allows entries in the bisimulation relation to encode information about what branch conditions they are under, thereby pruning some pair of paths that are simultaneously unreachable.

Recall from Section 6.3 that the constraints described above are the one that ensures the predicate at a pair of locations (gl_1, gl_2) imply that any visible instructions about to execute at (gl_1, gl_2) behave the same way. The visible instruction in this case are the parameterized statement variables like $\mathbf{S_1}$, and $\mathbf{S_2}$. The intuition behind choosing these statements as visible instructions is that since very little is known about them, we predict they should behave in the same way in the two programs.

Apart from the above kind of constraint there is another kind of constraint, which is used to state the relationship between one pair of related locations and other pairs of related locations. For example, if starting at (gl_1, gl_2) in states satisfying $\psi_{(gl_1,gl_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations (gl'_1, gl'_2), then $\psi_{(gl'_1,gl'_2)}$ must hold in the resulting states. As shown in Section 6.6.2 and again here in Section 7.6, such constraints can be stated over the constraint variables $\psi_{(gl_1,gl_2)}$ and $\psi_{(gl'_1,gl'_2)}$ using the weakest precondition operator (wp). This second kind of constraint guarantees that the computed bisimulation relation is in fact a bisimulation. In our example from Figure 7.4, the paths that our algorithm would discover between relation entries are as follows: A to B, B to E, B to C, C to D, D to C, D to E, and E to F.

During this step our algorithm also prune the infeasible paths. For example, when starting at E, it is impossible for the specification to stay in the loop – it must exit to F. Our algorithm can determine this from the predicate corresponding to the branch condition at E. In particular, let i and e be the original values of $\mathbf{I}$ and $\mathbf{E}$ at E (in either σ_1 or σ_2 since they are equal). The value e does not change through the loop as stated by the side conditions. If the original program chooses to stay in the loop, the `assume`$(\mathbf{I} < \mathbf{E})$ would lead to $i + 1 < e$ (where the "+1" comes from the increment of $\mathbf{I}$ and the fact that $\mathbf{S_2}$ does not modify $\mathbf{I}$). This would be inconsistent with the assumption from E stating that $i \geq e - 1$, and thus the path is pruned.

Solving Constraints

Once the constraints are generated, we solve them using an iterative algorithm that starts with all the constraint variables set to *true* and then iteratively strengthens the constraint variables until a theorem prover is able to show that all constraints are satisfied. Our algorithm first initializes the constraint variables with the conditions that are required for the visible instructions to be equivalent, thereby solving all the constraints of the first kind. Then it chooses any constraint of the second kind and iteratively solves it till it reaches a fix-point.

As an example, consider the constraint corresponding to the B-C path, our tool would ask a theorem prover to show that, for any σ_1 and σ_2: if (1) $\psi_{(a_2,b_1)} := (\sigma_1 = \sigma_2 \wedge \mathit{eval}(\sigma_1, \mathbf{I}) < \mathit{eval}(\sigma_1, \mathbf{E}))$ holds and (2) the original program executes $[\mathbf{S_1}]$ and (3) the transformed program executes $[\mathbf{S_1};\texttt{assume}(\mathbf{I} < \mathbf{E} - 1)]$, then $\psi_{(a_3,b_3)} := (\sigma_1 = \sigma_2 \wedge \mathit{eval}(\sigma_1, \mathbf{I}) < \mathit{eval}(\sigma_1, \mathbf{E}) \wedge \mathit{eval}(\sigma_2, \mathbf{I}) < \mathit{eval}(\sigma_2, \mathbf{E}) - 1)$ will hold after the two statements have executed. In this case, the implication follows immediately from the `assume` and the fact that $\mathbf{S_1}$ produces the same program state if started in the same program state. In fact for this example, the value of the constraint variable after solving the constraints of the first kind is indeed the bisimulation relation. If for some path pair X to Y, the implication does not hold (this is not the case in Figure 7.4), our algorithm would strengthen the current value of the constraint variable at X with the weakest precondition of the current value of the constraint variable at Y. Using such iterative strengthening, our algorithm tries to convert the original guessed relation into a bisimulation relation.

7.4 Parameterized Equivalence Checking

We now describe our approach in more detail. Our goal is to show that two parameterized programs P_1 and P_2 are equivalent under side conditions ϕ. We represent each program P as a transition diagram (Definition 5), which we denote by $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$. In particular, we assume that $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ is the transition diagram of the original program, and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ is the transition diagram of the transformed program. Moreover, we use ϵ_1, ϵ_2 to represent the exit locations of π_1, π_2.

We use a stronger definition of equivalence (redefined below) between two parameterized program, because parts of these programs are unknown. Furthermore, the programs we consider here are deterministic as such starting at a configuration $\langle \iota, \sigma \rangle$ there is at most one execution sequence that end in the exit location ϵ of the program.

Definition 33 (Final State)**.** *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$ and a state $\sigma \in \Sigma$, we use the notation $\pi(\sigma)$ to represent the final state after executing π starting in state σ. i.e.*

$$\pi(\sigma) = \sigma' \quad \textit{iff} \quad \exists\ \eta.\ \langle \iota, \sigma \rangle \ \overset{\eta}{\leadsto}^{+} \langle \epsilon, \sigma' \rangle$$

Definition 34 (Equivalence)**.** *Given two transition diagram π_1 and π_2, we define π_1 to be equivalent to π_2 if for any state $\sigma \in \Sigma$, we have $\pi_1(\sigma) = \pi_2(\sigma)$.*

The above definition of equivalence allows us to use our optimizations anywhere inside in a program: by establishing program state equivalence, we guarantee that the remainder of the program, after the optimization, runs the same way in the original and the transformed programs. We can model observable events such as IO using heap updates. For example, a call to `printf` can just append its arguments to a linked list on the heap. In this setting, our approach guarantees that the order of IO events is preserved.

The instructions in our transition diagram are taken from a concrete programming language of instructions, extended with meta-variables. The main components of our approach do not depend on the choice of the concrete language of instructions that we start with: this language can for example include pointers, arrays, and function calls. We do however make one exception to this rule: we assume the existence of `assume` instructions. In particular, we model conditionals using `assume` instructions on the transitions that flow away from a branch location (as shown for example in Figure 7.4). We also use `assume` instructions to insert the information from side conditions into the original or transformed program as needed, so that our tool can reason about the side conditions. The choice of concrete language only affects the semantics of instructions, which is entirely modularized in a function called *step* (which we have already seen). The only part of the system that knows about *step* is the theorem prover, which is given background axioms about the semantics of instructions (so that it knows for example how **I++** updates the store). All other parts of the system treat *step* as a black box.

Bisimulation Relation

Our approach is based on using a bisimulation relation to relate the execution of the original program and the transformed program. We use the same definitions as described in Section 6.4, however we redefine some of them again here to adapt them in the setting of parameterized programs. In our PEC system, we consider visible instructions to be the statement meta-variables (e.g. $\mathbf{S_1}, \mathbf{S_2} \in \vartheta$). We define two statement meta-variables to be equivalent if the variable name and the state of the program are the same. We also slightly modify the definition of simulation relation (Definition 22) to reflect the stronger definition of equivalence defined above.

Definition 35 (Simulation Relation)**.** *A simulation relation R for two transition dia-*

grams $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ *and* $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ *is a verification relation such that:*

$$R(\iota_1, \iota_2, \sigma_1 = \sigma_2) \text{ and } R(\epsilon_1, \epsilon_2, \sigma_1 = \sigma_2).$$

$$\begin{array}{l} \forall gl_2, gl'_2 \in \mathcal{L}_2, gl_1 \in \mathcal{L}_1, \sigma_1, \sigma_2, \sigma'_2 \in \Sigma, \phi \in \Phi, \eta_2 \in \mathcal{N}. \\ \quad \left[\begin{array}{l} \langle gl_2, \sigma_2 \rangle \overset{\eta_2}{\rightsquigarrow_2^+} \langle gl'_2, \sigma'_2 \rangle \wedge \\ \qquad\qquad R(gl_1, gl_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = true \end{array} \right] \Rightarrow \\ \quad\quad \exists gl'_1 \in \mathcal{L}_1, \sigma'_1 \in \Sigma, \phi' \in \Phi, \eta_1 \in \mathcal{N}. \\ \quad\quad\quad \left[\begin{array}{l} \langle gl_1, \sigma_1 \rangle \overset{\eta_1}{\rightsquigarrow_1^+} \langle gl'_1, \sigma'_1 \rangle \wedge \\ R(gl'_1, gl'_2, \phi') \wedge \phi'(\sigma'_1, \sigma'_2) = true \wedge \eta_1 \equiv \eta_2 \end{array} \right] \end{array}$$

The definition of bisimulation relation in this context is the same as in Definition 32. Furthermore, from Theorem 5 we know that if there exists a bisimulation relation between π_1 and π_2 then π_1 and π_2 are equivalent. Thus, a bisimulation relation is a witness that two transition diagrams are equivalent. Our approach is based on Theorem 5. In particular, our general approach is to try to infer a bisimulation relation to show that π_1 and π_2 are equivalent.

As before the GenerateConstraints and SolveConstraints module implement our bisimulation approach. These modules are similar to the one presented in Section 6.6. In particular, the GenerateConstraints module first generates a set of pair-of-interest locations $\mathcal{R}$ from the two transition diagrams π_1 and π_2, and then generates a set of constraints $\mathcal{C}$. The SolveConstraints module next solves these constaints such that the properties from Definitions 35 and 32 hold, possibly strengthening the relation in order to guarantee property 3 of Definitions 35. The next two sections describe these modules of our system.

7.5 GenerateConstraints Module

To prove that two parameterized programs are equivalent our approach attempts to discover a bisimulation relation between them. To do this, the GenerateConstraints module computes a set of constraints for the set of pair-of-interests, which will then be strengthened to a bisimulation relation by the SolveConstraints module.

Here again to focus our attention on only those locations for which our approach infers the relation entries, we define two sets of locations $\mathcal{Q}_1$ and $\mathcal{Q}_2$ for the transition diagrams π_1 and π_2 respectively. These are the locations that immediately precede a statement meta-variable. We use the *skipping transition* relations $gl_1 \overset{(w_1,\mathcal{Q}_1)}{\hookrightarrow}_{\pi_1} gl'_1$ and $gl_2 \overset{(w_2,\mathcal{Q}_2)}{\hookrightarrow}_{\pi_2} gl'_2$ (Definition 23) that skips over all locations not in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ respectively. We also use the definition of *parallel transition* relation $\hookrightarrow\!\!\!\rightarrow$ (Definition 26) that essentially traverses the two transition diagrams (specification and implementation) in synchrony, while focusing on only those locations that are in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ respectively.

The predicate $\mathsf{Rel} : \mathcal{I}^* \times \mathcal{I}^* \times \mathcal{Q}_1 \times \mathcal{Q}_2 \rightarrow \mathcal{B}$ used in the parallel transition definition checks if the pair of paths are feasible. To do this we conservatively compute if two paths are *infeasible*, and if not then we say they are feasible i.e.

$$\mathsf{Rel}(w_1, w_2, gl_1, gl_2) = \neg\mathsf{Infeasible}(w_1, w_2, gl_1, gl_2)$$

For infeasiblity check, we first define two sets of locations $\mathcal{A}_1$ and $\mathcal{A}_2$ for π_1 and π_2 respectively. These sets consists of locations that immediately precede an assume statement. We define for each pair $(gl_1, gl_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$ a variable $\mathcal{X}_{(gl_1,gl_2)}$ such that:

$$\begin{aligned} \mathcal{X}_{(gl_1,gl_2)} &= \mathit{Post}(gl_1, \mathcal{A}_1, \pi_1) \wedge \mathit{Post}(gl_2, \mathcal{A}_2, \pi_2) \wedge \sigma_1 = \sigma_2 \quad \text{and} \\ \mathit{Post}(gl, \mathcal{A}, \pi) &= \bigvee_{\{gl' \overset{(w,\mathcal{A})}{\hookrightarrow}_{\pi} gl\}} \mathsf{sp}(w, \mathit{true}) \end{aligned}$$

Here $\mathcal{X}_{(gl_1,gl_2)}$ computes a conservative formula over σ_1 and σ_2 that should hold when π_1 and π_2 are at locations gl_1 and gl_2 respectively. Within $\mathcal{X}$, the predicate $\mathit{Post}(gl, \mathcal{A}, \pi)$ is the disjunction of the strongest post conditions with respect to *true* over paths w for which there exists some gl' such that $gl' \overset{(w,\mathcal{A})}{\hookrightarrow}_{\pi} gl$.

Our implementation of the Infeasible function can be succinctly represented as follows:

$$\mathsf{ATP}(\neg(\mathsf{sp}(w_1, \mathcal{X}_{(gl_1,gl_2)}) \wedge \mathsf{sp}(w_2, \mathcal{X}_{(gl_1,gl_2)}))) = \mathit{Valid}$$

Infeasible first computes the strongest postcondition of w_1 and w_2 with respect to the formula $\mathcal{X}_{(gl_1,gl_2)}$. If an automated theorem prover (ATP) can show that the two postconditions are inconsistent, then the combination of those two paths is infeasible, and can be pruned. The Infeasible function performs the pruning that was intuitively described for the loop pipelining example in Section 7.3.2.

The other predicate $\mathsf{WellMatched} : \mathcal{I}^* \times \mathcal{I}^* \rightarrow \mathcal{B}$ in the definition of parallel transition $\hookrightarrow\!\!\!\rightarrow$ checks if the two sequences w_1 and w_2 of instructions are well-matched i.e. neither of them contain a statement meta-variable or they each contains the same statement meta-variable. Using these definitions of Rel, WellMatched and relation $\hookrightarrow\!\!\!\rightarrow$, we now use the relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ of location pairs that will form the entries of our bisimulation relation (see Definition 27). As before the set $\mathcal{R}$ can easily be computed by starting with the empty set, and applying the three rules exhaustively. We then check if the computed set $\mathcal{R}$ is a well-formed pairs of interest relation using Definition 28.

To uniformly handle the side conditions ϕ, we insert the side condition assumptions into the original and transformed programs in the form of `assume` statements. An `assume` statement takes as argument a predicate over the program state σ that occurs at the point where the assume holds. To ease presentation, we make the simplifying assumption that $\phi = \phi_1 @ L_1 \wedge \ldots \wedge \phi_n @ L_n$ (our implementation handles the general case). For each side condition ϕ_i, we define $[\![\phi_i]\!]$ to be a predicate over σ that directly encodes the side condition's meaning provided by the optimization writer. Then for each $\phi_i @ L_i$, we find the location L_i in either the original or the transformed program, and insert `assume`($[\![\phi_i]\!]$) at that location.

Similar to Section 6.6.2, we now describe our PEC algorithm in terms of constraint solving. In particular, for each $(gl_1, gl_2) \in \mathcal{R}$ we define a constraint variable $\psi_{(gl_1, gl_2)}$ (Definition 29) representing the predicate that we want to compute for the bisimulation entry (gl_1, gl_2). Using these constraint variables, the final bisimulation relation will have the form:

$$\{(gl_1, gl_2, \psi_{(gl_1, gl_2)}) \mid \mathcal{R}(gl_1, gl_2)\}$$

To compute the predicates that the constraint variables $\psi_{(gl_1, gl_2)}$ stand for, we generate a set of constraints on these variables, and then solve the constraints. We use a slightly modified version of Definition 30 to compute the set of constraints $\mathcal{C}$.

Definition 36 (Set of Constraints)**.** *The set $\mathcal{C}$ of constraints is defined by:*

$\left[\psi_{(\iota_1, \iota_2)} \Rightarrow \sigma_1 = \sigma_2\right] \in \mathcal{C}$ *and* $\left[\psi_{(\epsilon_1, \epsilon_2)} \Rightarrow \sigma_1 = \sigma_2\right] \in \mathcal{C}$

For each (gl_1, gl_2) *in* $\mathcal{R}$*:* $\left[\psi_{(gl_1, gl_2)} \Rightarrow \mathcal{X}_{(gl_1, gl_2)}\right] \in \mathcal{C}$

For each $(gl_1, gl_2) \overset{(w_1, w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2)$*:* $\left[\psi_{(gl_1, gl_2)} \Rightarrow \mathsf{pwp}(w_1, w_2, \psi_{(gl'_1, gl'_2)})\right] \in \mathcal{C}$

There are three kinds of constraints. The first kind of constraints make sure that if the two parameterized programs start in equal states then they end in equal states. The next kind of constraints implies that for each pair of locations (gl_1, gl_2) the instructions about to execute at gl_1 and gl_2 are equivalent. This condition in our PEC setting is captured using the formula $\mathcal{X}_{(gl_1, gl_2)}$.

The last kind of constraints state that for each pair of paths between two entries in $\mathcal{R}$, the predicate at the beginning of the paths must imply the predicate at the end of the paths. We express this condition using the weakest precondition computation pwp, which is a parameterized version of the regular weakest precondition.

The main challenge in expressing this weakest precondition is that the traditional formulation of weakest precondition depends on the structure of the statements being processed. As a result, it is difficult to use this definition for statements like $\mathbf{S_1}$ and $\mathbf{S_2}$ in our parameterized programs, because the precise structure of these statements is not known. To address this challenge, we use an alternate yet equivalent definition of weakest precondition. In particular, consider the traditional weakest precondition computation, and assume that the predicate we are computing is a function from program states to booleans. Then the traditional weakest precondition wp can be expressed as:

$$\mathsf{wp}(\mathbf{S}, \varphi)(\sigma) = \varphi(\mathit{step}(\sigma, \mathbf{S}))$$

If we assume that the program state σ is simply a free variable in the predicate φ, then wp can be expressed as:

$$\mathsf{wp}(\mathbf{S}, \varphi) = \varphi[\sigma \mapsto \mathit{step}(\sigma, \mathbf{S})]$$

Generalizing this to two parallels paths in two different programs, the predicates now have free variables σ_1 and σ_2, and we can express pwp as follows:

$$\mathsf{pwp}(w_1, w_2, \varphi) = \varphi[\sigma_1 \mapsto \mathit{step}(\sigma_1, w_1), \sigma_2 \mapsto \mathit{step}(\sigma_2, w_2)]$$

7.6 SolveConstraints Module

Once the set of constraints $\mathcal{C}$ have been generated, the SolveConstraints module tries to solve these constraints iteratively by starting with the all the *constraint variables* initialized to *true*, and iteratively strengthening the constraint variables in the relation until all the constraints are satisfied. The SolveConstraints function used here is the exact function presented in Figure 6.6. As before one subtlety is that we cannot strengthen the relation at the entry points ι_1, ι_2. If we ever try to do this, we indicate an error. Here again because SolveConstraints is trying to compute a fixed-point over the very flexible but infinite domain of boolean formulas, it may not terminate. However, in our experiments we found that in practice SolveConstraints can quickly find a fix-point.

7.7 Experiments and Results

We implemented our PEC algorithm using the Simplify theorem prover [DNS05] to realize the ATP module from Section 7.5. Using our language we expressed various transformations and proved them correct using the PEC tool. The transformations proved correct by the PEC tool are: copy propagation, constant propagation, common sub-expression elimination, partial redundancy elimination, loop invariant code hoisting, conditional speculation, software pipelining, loop unswitching, loop unrolling, loop peeling, and loop splitting.

The trusted computing base for our system includes: (1) the PEC checker, comprising 2,408 lines of OCaml code (2) the Simplify automated theorem prover, a widely used and well tested theorem prover, and (3) the execution engine that will run the optimizations. Within the execution engine, the trust can be further subdivided into two components. The first component of the execution engine must perform the syntactic pattern matching for rewrite rules, and apply rewrite rules when they fire. This part is always trusted. The second component of the execution engine must perform program analysis to check each optimization's side-conditions in a way that guarantees their semantic meaning. Here our system offers a choice. These analysis can either be trusted and thus implemented inside the compiler using arbitrarily complex analysis, or untrusted and implemented using a provably safe analysis system like Rhodium.

7.8 Related work

Our work presented here is related to long lines of work in translation validation, proving loop optimizations correct, automated correctness checking of optimizations, human-assisted correctness checking of optimizations, and languages for expressing optimizations. We now discuss each area in more detail.

Translation Validation: Our approach is heavily inspired by the work that has been done on translation validation [PSS98, Nec00, RM99, GZB05, KLG07, KLG08]. However, unlike previous translation validation approaches, our equivalence checking algorithm addresses the challenge of reasoning about statements that are not fully specified. As a result, our approach is a *generalization* of previous translation validation techniques that allows optimizations to be proved correct once and for all. Furthermore, because our PEC approach can handle concrete statements as well as parameterized statements, it subsumes many of the previous approaches to translation validation.

Automated correctness checking of optimizations: As with our PEC algorithm, the Cobalt [LMC03] and Rhodium [LMRC05] systems are able to check the correctness of optimizations once and for all. However, Cobalt and Rhodium only support rewrite rules that transform a single statement to another statement, thus limiting the kinds of optimizations they can express and prove correct. Our PEC approach can handle complex many-to-many rewrite rules explicitly, allowing it to prove many more optimizations correct.

Human-assisted correctness checking of optimizations: A significant amount of work has been done on manually proving optimizations correct, including abstract interpretation [CC77, CC02], the work on the VLISP compiler [GRW95], Kleene algebra with tests [Koz97], manual proofs of correctness for optimizations expressed in temporal logic [Ste91, LJWF02], and manual proofs of correctness based on partial equivalence relations [Ben04]. Analyses and transformations have also been proven correct mechanically, but not automatically: the soundness proof is performed with an interactive theorem prover that requires guidance from the user. For example, Young [You89] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [KB95]. As another example, Cachera *et al.* [CJPR04] show how to specify static analyses and prove them correct in constructive logic using the Coq proof assistant. Via the Curry-Howard isomorphism, an implementation of the static analysis algorithm can then be extracted from the proof of correctness. Leroy's Comcert project [Ler06] has also used a similar technique to manually develop a semantics preserving, optimizing compiler for a large subset of C. The Comcert compiler provides an end-to-end correctness guarantee, and does not just focus on optimizations, as we do in our approach. Tristan *et al.* has also proved that certain translation validators are correct once and for all, but here again by implementing the proof manually [TL08, TL09]. In all these cases, however, the proof requires help from the user. In contrast to these approaches, our proof strategy is fully automated but trusts that the side conditions are computed correctly when the compiler executes.

Languages for expressing optimizations: The idea of analyzing optimizations written in a specialized language was introduced by Whitfield and Soffa with the Gospel language [WS97]. Many other frameworks and languages have been proposed for specifying dataflow analyses and transformations, including Sharlit [TH92], System-Z [YHI93], languages based on regular path queries [SdML04], and temporal logic [Ste91, LJWF02]. None of these approaches addresses automated correctness checking of the specified optimizations.

7.9 Summary

In this chapter we presented Parameterized Equivalence Checking (PEC), a technique for automatically proving optimizations correct once and for all. PEC bridges the gap between translation validation and once-and-for-all techniques. Our PEC approach generalizes previous translation validation techniques to handle parameterized programs, which are partially specified programs that can represent multiple concrete programs, thereby adapting them to provide once and for all correctness proofs. Furthermore, our use of expressive many-to-many rewrite rules and a robust proof technique enables PEC to automatically prove correct optimizations that have been difficult or impossible to prove in previous systems. We have also implemented our PEC algorithm and proved a variety of optimizations correct.

Acknowledgements: Chapter 7, in part, have been accepted for publication as “Proving Optimizations Correct using Parameterized Program Equivalence” by Sudipta Kundu, Zachary Tatlock, and Sorin Lerner in *PLDI 09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation* [KTL09]. The dissertation author was the primary investigator and author of the parts of this paper that is included in this dissertation.

Chapter 8

Conclusions and Future Work

We have addressed the need for high-level verification methodologies that allows us to do functional verification early in the design phase and then iteratively use correct refinement steps to generate the final RTL design. We believe that by performing verification on the high-level design, where the design description is smaller in size and the design intent information is easier to extract, and then checking that all refinement steps are correct, the domain of high-level verification can provide strong and expressive guarantees that would have been difficult to achieve by directly analyzing the low-level RTL code.

The high-level verification methods can be broadly seen as methods for verifying properties of high-level designs and methods for verifying that the translation from high-level design to low-level RTL preserves semantics. We classified the high-level verification area into three main parts, namely high-level property checking, translation validation, and synthesis tool verification. In this thesis we have explored techniques in each of the above three areas.

High-Level Property Checking

For high-level property checking, we use model checking techniques to verify that a design satisfies a given property such as absence of deadlocks or assertion violations. In particular, we explored two techniques one on execution-based model checking and the other on symbolic model checking.

We implemented Satya, an execution-based model checking tool that combines static and dynamic POR techniques along with high-level semantics of SystemC to intelligently explore all possible behaviors of a SystemC design. Our approach reduces the runtime overhead by conservatively computing the dependency information statically and using it during runtime, without significant loss of precision. In our experiments Satya was able to automatically find an assertion violation in the FIFO benchmark (distributed as a part of the OSCI repository), which may not have been found by simulation (Section 4.10).

We also developed Candor, a symbolic analysis tool for concurrent C programs that combines POR with a previous asynchronous modeling approach called token-passing approach. The token-passing approach generates verification conditions directly without an explicit scheduler, thereby avoiding some of the limitations of synchronous modeling approach. However, this approach add interleaving constraints between all pairwise global accesses, thereby allowing redundant interleavings. To address this we introduce the notion of *Mutually Atomic Transactions* (MAT): two transactions are mutually atomic when there exists exactly one conflicting shared-access pair between them. Using MATs, we then reduce the verification conditions by allowing pairwise interleaving constraints *only* between MATs. Our experimental results show that our approach improves the current state of the art both in performance and in size of the verification condition (Section 5.10).

Translation Validation

To verify the translation from the high-level design to low-level RTL is correct, we developed a translation validation tool called Surya. Our algorithm uses a *bisimulation relation* approach to automatically prove the equivalence between two concurrent systems represented as transition diagrams. We used Surya to validate the synthesis process of Spark, a parallelizing HLS framework. Surya validates all the phases (except for parsing, binding and code generation) of Spark against the initial behavioral description. Our experiments showed that with only a fraction of the development cost of Spark, Surya can validate the translations performed by Spark, and it even uncovered two previously unknown bugs that eluded long-term use.

Synthesis Tool Verification

For synthesis tool verification, we developed a technique that proves the correctness of optimizations using *Parametrized Equivalence Checking* (PEC). Our approach proves the correctness of the optimizations *once and for all*, before it is ever run. The PEC technique is a generalization of translation validation that proves the equivalence of *parameterized programs*. To highlight the power of PEC, we designed a language for implementing complex optimizations using many-to-many rewrite rules, and used this language to implement a variety of optimizations including software pipelining, loop unrolling, and loop unswitching. Using our PEC implementation, we were able to automatically verify that all the optimizations we implemented in our language preserve program behavior.

Future Work

In this thesis we focused on exploring techniques in the area of high-level verification. Recent advances in formal methods and HLS have invigorated interest in high-level verification both in industry and academics. Various verification tools and techniques focused toward high-level design are starting to emerge. However, their adoption is in the early stages and the tools are often limited in the quality of the results and the kinds of correctness guarantees that are provided. Naturally there are many things to be done in this area. In this section we discuss promising future research areas in verification of high-level designs and the tools associated with them.

Hardware-Software modeling: High-level hardware languages support many features that are useful for both software and hardware designs. For example, SystemC allows both asynchronous and synchronous semantics of concurrency, and also both software and hardware data types. However, existing symbolic analysis tools including ours often either target software or hardware. For example, most software model checkers only support software data types and asynchronous semantics of concurrency, and most hardware model checker only support hardware data types and synchronous semantics of concurrency. As a result, researchers often use abstraction or complicated techniques while modeling the non-supported features of a given model checker. This gap points to a possible research direction that would unify techniques for hardware models and techniques for software models into combined methodology for reasoning about hardware-software models.

Compositional techniques: Although many techniques presented in this thesis use compositional methods to make the verification problem tractable, these techniques are still limited in their application. Even after decomposition using the current techniques the problem is still quite large and complex. Advanced and more efficient methods are needed for decomposing a computationally demanding global property into local properties whose verification is simpler.

Modular framework: One observation of this thesis is that there are variety of overlap between various verification techniques. However, each verification tool is highly tuned toward the particular problem it is solving with its own input language or API, and as such it is hard to modify or extend. Hence, every time a new methodology is proposed a new tool has to be written, often from scratch. Unfortunately, the tools discussed in this thesis are not designed from a software engineering perspective. Thus, there is a need for a modular and reusable framework, which can be quickly used to prototype new ideas and test them.

Debugging: The tools discussed in this thesis except Candor provide only limited feedback to the user. When a bug is found, these tools cannot typically pin-point the error in the code. All the methods are able to output an error trace, but figuring out the cause of the error from it, is not straightforward and requires expertise in formal methods. Although not directly related to high-level verification, there has been work in this area [BNR03], however adapting such techniques to the this domain is still a challenge. Another limitation of our methods is that they often stop searching when a bug is found, rather than providing a list of all bugs. More broadly, the goal should be to fit formal verification into the regular develop-edit-debug flow, which would require the development of verification tools for speed and ease of use.

Synthesis-For-Verification: HLS process focuses mainly on three design constraints: area, timing and power. These methodologies tend to ignore verification, which takes about 70% of the design cycle, as a constraint. Recently, Ganai et al. [GMGW07] proposed a new paradigm ‘Synthesis-For-Verification’ which involves synthesizing “verification-aware” designs that are more suitable for functional verification. Therefore, another research direction may be to use existing infrastructure of HLS to generate “verification friendly” models that are relatively easier to verify using state-of-the-art techniques.

Compiler techniques: Many techniques used in HLS are similar to those used for compilers. As a result, advances in fields like compiler correctness can provide inspiration for developing techniques for high-level verification. For example, our translation validation and synthesis tool verification work are inspired from the work done in the area of compiler correctness such as Necula's translation validation technique [Nec00], Zuck et al. [GZB05] work on proving various non-structure preserving transformation, and Lerner et al. [LMC03, LMRC05] approach to automatically prove the correctness of compiler optimizations once and for all. There are many other techniques that have been successfully applied to the compiler domain, and can provide new directions for verification of the HLS process.

Bibliography

[ABH$^+$01] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods System Design*, 18(2):97–116, 2001.

[ABRM98] Pranav Ashar, Subhrajit Bhattacharya, Anand Raghunathan, and Akira Mukaiyama. Verification of RTL generated from scheduled behavior in a high-level synthesis flow. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*, pages 517–524, 1998.

[AHMN91] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Computer Architecture News*, 19(3):234–243, 1991.

[AQR$^+$04] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 04: 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 1–15. Springer Verlag, 2004.

[ARGB99] Pranav Ashar, Anand Raghunathan, Aarti Gupta, and Subhrajit Bhattacharya. Verification of scheduling in the presence of loops using uninterpreted symbolic simulation. In *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 458–466, Washington, DC, USA, 1999. IEEE Computer Society.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 12–21, New York, NY, USA, 2007. ACM.

[BBC$^+$06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, New York, NY, USA, 2006. ACM.

[BCC$^+$99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320, New York, NY, USA, 1999. ACM.

[BCM$^+$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.

[BDP00] Dominique Borrione, Julia Dushina, and Laurence Pierre. A compositional model for the functional verification of high-level synthesis results. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):526–530, 2000.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, January 2004.

[BG00] Doran Bustan and Orna Grumberg. Simulation based minimization. In David A. McAllester, editor, *Proceedings of the International Conference on Automated Deduction*, volume 1831 of *LNCS*, pages 255–270. Springer Verlag, 2000.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[Bla00] Claudia Blank. Formal verification of register binding. In *WAVE '00: Proceedings of the Workshop on Advances in Verification*, 2000.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the 2001 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2001.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, pages 97–105, 2003.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Princi-*

ples of Programming Languages, pages 238–252, Los Angeles CA, January 1977.

[CC02] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.

[CCG03] Sagar Chaki, Edmund Clarke, and Alex Groce. Modular verification of software components in C. In *IEEE Transactions on Software Engineering*, pages 385–395, 2003.

[CD93] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[CG03] Lukai Cai and Daniel Gajski. Transaction Level Modeling: an overview. In *Proceedings of International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2003.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Cha88] K. Mani Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[CJPR04] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Proceedings of the 13th European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In *SPIN '05: Proceedings of the 12th international workshop on Model Checking Software*, pages 75–90, 2005.

[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.

[CLM89] Edmund M. Clarke, David E. Long, and Ken. L. McMillan. Compositional model checking. Technical Report CMS-CS-89-145, School of Computer Science, Canergie Mellon University, April 1989.

[Cor] IBM Corporation. Rational Statemate. `www.telelogic.com/products/statemate`.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of the Workshop on Software Model Checking (SoftMC)*, volume 89 of *ENTCS*, 2003.

[DBJ98] Julia Dushina, Dominique Borrione, and Ahmed Amine Jerraya. Formal verification of the allocation step in high level synthesis. In *Forum on Design Languages (FDL'98)*, Lausanne, Switzerland, 1998.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 160–171, London, UK, 1999. Springer-Verlag.

[Dil96] David L. Dill. The murphi verification system. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag.

[DNS05] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the Association for Computing Machinery*, 52(3):365–473, May 2005.

[DS97] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

[EBK96] Dirk Eisenbiegler, Christian Blumenröhr, and Ramayya Kumar. Implementation issues about the embedding of existing high level synthesis algorithms in hol. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 157–172, London, UK, 1996. Springer-Verlag.

[(ED92] Edison Design Group (EDG). C/C++ Front End, 1992. `www.edg.com`.

[EHR99] Hans Eveking, Holger Hinrichsen, and Gerd Ritter. Automatic verification of scheduling results in high-level synthesis. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 12, New York, NY, USA, 1999. ACM Press.

[EP] EP2: Electronic Payment 2. `www.eftpos2000.ch`.

[ES96] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.

[FLL$^+$02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2002.

[FQ03] C. Flanagan and S. Qadeer. Transactions for software model checking. *Electronic Notes in Theoretical Computer Science*, 89, 2003.

[FV99] Kathi Fisler and Moshe Y. Vardi. Bisimulation and Model Checking. In *Proceedings of the 10th Conference on Correct Hardware Design and Verification Methods*, Bad Herrenalb Germany CA, September 1999.

[GB08] Rajesh Gupta and Forrest Brewer. High-Level Synthesis: A Retrospective. In *High-Level Synthesis from Algorithm to Digital Circuit*. Springer, 2008.

[GDGN03] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *International. Conference on VLSI Design*, 2003.

[GDWL92] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[GED07] Daniel Grobe, Rüdiger Ebendt, and Rolf Drechsler. Improvements for constraint solving in the SystemC verification library. In *GLSVLSI '07: Proceedings of the 17th ACM Great Lakes symposium on VLSI*, pages 493–496, New York, NY, USA, 2007. ACM.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *SPIN '07: Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 95–112, 2007.

[GG06] Malay K Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pages 794–801, New York, NY, USA, 2006. ACM.

[GG08] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in bmc. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 114–133, Berlin, Heidelberg, 2008. Springer-Verlag.

[GK09] Malay Ganai and Sudipta Kundu. Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In *SPIN '09: Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, 2009.

[GL97] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design and Test*, 14(2):72–80, 1997.

[GLMS02] T. Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GLST05] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 122–131, New York, NY, USA, 2005. ACM.

[GMGW07] Malay K Ganai, Akira Mukaiyama, Aarti Gupta, and Kazutoshi Wakabayshi. Synthesizing "verification aware" models: Why and how? *VLSI Design '07: 20th International Conference on VLSI Design*, 0:50–56, 2007.

[GN07] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[God95] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* PhD thesis, Univerite De Liege, 1995.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Gor88] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[GP92] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transaction on Parallel Distributed Systems*, 1992.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 438–449, London, UK, 1993. Springer-Verlag.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[GRW95] J. Guttman, J. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1-2):33–110, 1995.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs of infinite systems with PVS. In *CAV '97: Proceedings of the international conference on Computer Aided Verification*, June 1997.

[Gup92] Aarti Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, 1(2-3):151–238, 1992.

[GZB05] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, May 2005.

[HDL89] F. K. Hanna, N. Daeche, and M. Longley. Formal Synthesis of Digital Systems. In L. Claesen, editor, *Proc IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 532–548. Elsevier, 1989. Leuven, Belgium.

[HDZ00] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher Order Symbolic Computation*, 13(4):315–353, 2000.

[Hin98] H. Hinrichsen. Language of labelled segments documentation. Technical report, Darmstadt University of Technology, 1998.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, January 2002.

[HMMCM06] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HT05] Ali Habibi and Sofiene Tahar. Design for verification of SystemC Transaction Level Models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565, Washington, DC, USA, 2005. IEEE Computer Society.

[Ini05] Open SystemC Initiative. IEEE Standard 1666 SystemC Language Reference Manual, 2005. `www.systemc.org`.

[IR05] Yoshinao Isobe and Markus Roggenbach. A generic theorem prover of CSP refinement. In *TACAS '05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 103–123. Springer-Verlag, April 2005.

[IYG$^+$08] Franjo Ivanicic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[JB91] S. Johnson and B. Bose. DDD — a system for mechanized digital design derivation. In *ACM/SIGDA Workshop on Formal Methods in VLSI Design*, Miami, Florida, jan 1991.

[Jos88] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3(1):9–18, March 1988.

[Kah01] Andrew B. Kahng. Design technology productivity in the DSM era (invited talk). In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 443–448, New York, NY, USA, 2001. ACM.

[KB95] M. Kauffmann and R.S. Boyer. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, January 1995.

[KBP07] Alfred Koelbl, Jerry R. Burch, and Carl Pixley. Memory modeling in ESL-RTL equivalence checking. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 205–209, New York, NY, USA, 2007. ACM.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[KGG08] Sudipta Kundu, Malay Ganai, and Rajesh Gupta. Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In *DAC '08: Proceedings of the 45th annual conference on Design Automation*, pages 936–941, New York, NY, USA, 2008. ACM.

[KGS06] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. Symbolic Model Checking of Concurrent Programs Using Partial Orders and On-the-Fly Transactions. In *CAV '06: Proceedings of the 18th international conference on Computer Aided Verification*, pages 286–299, 2006.

[KKM04] Youngsik Kim, Shekhar Kopuri, and Nazanin Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In *ISQED '04: Proceedings of the 5th International Symposium on Quality Electronic Design*, pages 110–115, Washington, DC, USA, 2004. IEEE Computer Society.

[KLG07] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Automated Refinement Checking of Concurrent Systems. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 318–325, Piscataway, NJ, USA, 2007. IEEE Press.

[KLG08] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Validating High-Level Synthesis. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 459–472, Princeton, NJ, USA, 2008. Springer.

[KLG09] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. High-Level Verification. *IPSJ Transactions on System LSI Design Methodology*, 2009. (Invited Paper).

[KMS$^+$06] C Karfa, C Mandal, D Sarkar, S R Pentakota, and Chris Reade. A formal verification method of scheduling in high-level synthesis. *IEEE International Symposium on Quality Electronic Design*, 0:71–78, 2006.

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Langauges and Systems*, 19(3):427–443, September 1997.

[KS05] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *Proceedings of Third ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2005.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving Optimizations Correct using Parameterized Program Equivalence. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[Lam88] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the 1988 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 1988.

[Lar96] Mats Larsson. Improving the result of high-level synthesis using interactive transformational design. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 299–314, London, UK, 1996. Springer-Verlag.

[LBC05] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, pages 24–38, 2005.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, 2006.

[Lin97] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems.*, 2(1):2–21, 1997.

[LJWF02] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, January 2002.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the 2003 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2003.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.

[LPQR05] Vladimir Levin, Robert Palmer, Shaz Qadeer, and Sriram K. Rajamani. Sound transaction-based reduction without cycle detection. In *SPIN '05: Proceedings of the 12th international workshop on Model Checking Software*, pages 106–122, 2005.

[LR08] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 37–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[LSV98] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[Ltd] Formal Systems (Europe) Ltd. Failures-divergence refinement: FDR2 user manual. Oxford, England, June 2005.

[Maz87] A Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[MB08] Leonardo De Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[McF93a] M. C. McFarland. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):654–663, 1993.

[McF93b] Michael C. McFarland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2(3):231–257, 1993.

[McM00] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[MHMP02] J. M. Mendías, R. Hermida, M. C. Molina, and O. Penalba. Efficient verification of scheduling, allocation and binding in high-level synthesis. In *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 308, Washington, DC, USA, 2002. IEEE Computer Society.

[Mic90] Giovanni De Micheli. Guest editorial: High-level synthesis of digital circuits. *IEEE Transactions on Design Test*, 7(5):6–7, 1990.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[Mic05] ST Microelectronics. Transaction Accurate Communication (TAC) platform, 2005. `www.greensocs.com/TACPackage`.

[MMMC05] M. Moy, Maraninchi, and Maillet-Contoz. Lussy: A toolbox for the analysis of Systems-on-a-Chip at the Transactional Level. In *Proceedings of International Conference on Application of Concurrency to System Design (ACSD)*, 2005.

[MMZ$^+$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.

[Moo98] J. Strother Moore. Symbolic simulation: An acl2 approach. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 334–350, 1998.

[MPC$^+$02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[Mus08] Madan Musuvathi. Systematic concurrency testing using chess. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–1, New York, NY, USA, 2008. ACM.

[MV00] Nazanin Mansouri and Ranga Vemuri. Automated correctness condition generation for formal verification of synthesized RTL designs. *Formal Methods in System Design: An International Journal*, 16(1):59–91, January 2000.

[MVD03] Samy Meftali, Jol Vennin, and Jean-Luc Dekeyser. A fast SystemC simulation methodology for Multi-Level IP/SoC design. In *IFIP International Workshop on IP Based SoC Design*, 2003.

[Nar98] Naren Narasimhan. *Theorem Proving Guided Development of Formal Assertions and their embedding in a High-Level VLSI Synthesis System*. PhD thesis, University of Cincinnati, 1998.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the 2000 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2000.

[NTR$^+$01] Naren Narasimhan, Elena Teica, Rajesh Radhakrishnan, Sriram Govindarajan, and Ranga Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Formal Methods in System Design*, 19(3):237–273, 2001.

[NV98] Naren Narasimhan and Ranga Vemuri. On the effectiveness of theorem proving guided discovery of formal assertions for a register allocator in a high-level synthesis system. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, pages 367–386, Canberra, Australia, 1998. Springer-Verlag.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[Pau94] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecure Notes in Computer Science*. Springer Verlag, 1994.

[Pel98] D. Peled. Ten years of partial order reduction. In *CAV '98: Proceedings of the international conference on Computer Aided Verification*, June 1998.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS '05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2005.

[Raj95] Sreeranga P. Rajan. Correctness of transformations in high level synthesis. In Steven D. Johnson, editor, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, pages 597–603, Chiba, Japan, 1995.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programing Language Systems*, 22(2):416–430, 2000.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *CAV '05: Proceedings of the 17th international conference on Computer Aided Verification*, pages 82–97, 2005.

[RGG$^+$95] A. W. Roscoe, Paul H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 1020 dining philosophers for deadlock. In *TACAS '95: Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 133–152, London, UK, 1995. Springer-Verlag.

[RM99] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.

[RTV00] R. Radhakrishnan, E. Teica, and R. Vermuri. An approach to high-level synthesis system validation using formally verified transformations. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 80, Washington, DC, USA, 2000. IEEE Computer Society.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[SC06] Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. *Formal Methods in System Design*, 28(3):263–289, 2006.

[SDGK07] Rudrapatna Shyamasundar, Frederic Doucet, Rajesh Gupta, and Ingolf Kruger. Compositional reactive semantics of SystemC and verification with RuleBase. In *Proceedings of the GM R&D Workshop*, 2007.

[SdML04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice Italy, January 2004.

[SG07] Alper Sen and Vijay K. Garg. Formal verification of simulation traces using computation slicing. *IEEE Transactions on Computers*, 2007.

[SJ04] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[SR97] Robin Sharp and Ole Rasmussen. The T-Ruby design system. *Formal Methods in System Design: An International Journal*, 11(3):239–264, October 1997.

[SRI] SRI. An SMT solver. `http://fm.csl.sri.com/yices/`.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Science (TACS), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364. Springer-Verlag, September 1991.

[Swa06] Stuart Swan. Systemc transaction level models and rtl verification. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 90–92, New York, NY, USA, 2006. ACM.

[Sys] Calypto Design Systems. Slec system. `www.calypto.com/slecsystem.php`.

[Tec] Esterel Technologies. Scade design verifier. `www.esterel-technologies.com/`.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit – A tool for building optimizers. In *PLDI '92: Proceedings of the 1992 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, 2008.

[TL09] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2009.

[TVGSV95] Robert K. Brayton The VIS Group and Alberto L. Sangiovanni-Vincentelli. Vis: A system for verification and synthesis. Technical Report UCB/ERL M95/104, EECS Department, University of California, Berkeley, 1995.

[TW97] Haykal Tej and Burkhart Wolff. A corrected failure divergence model for csp in isabelle/hol. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, pages 318–337, London, UK, 1997. Springer-Verlag.

[Var07] Moshe Y. Vardi. Formal techniques for SystemC verification. In *DAC '07: Proceedings of the 44th annual conference on Design Automation*, 2007.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12. Press, 2000.

[WC91] R. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, Boston, MA, USA, 1991.

[WS97] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[WYKG08] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS '08: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396, 2008.

[YGL04] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Memory-model-sensitive data race analysis. In *ICFEM '04: Proceedings of 6th International Conference on Formal Engineering Methods*, pages 30–45, 2004.

[YGLS03] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[YHI93] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[You89] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.

[ZPG$^+$05] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.