The Dissertation Committee for Shobha Vasudevan
certifies that this is the approved version of the following dissertation:

High Level Static Analysis of System Descriptions for Taming Verification Complexity

Committee:

Jacob A. Abraham, Supervisor

E. Allen Emerson

Adnan Aziz

Jason Baumgartner

Vijay Garg

High Level Static Analysis of System Descriptions for Taming Verification Complexity

by

Shobha Vasudevan, B.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2007

To Swami, for witnessing the process...

To Amma and Daddy, for every step, from then to now...

To Siddarth, Vinod, and Sumant for it belongs to them.

Acknowledgments

At the end of a lifetime experience, it is cathartic to look back and write an acknowledgement. It overwhelms and stings, exhilarates and enervates, causes righteous pride and knowing guilt, convicts and liberates- all at once. Every PhD has a compelling story, and i feel urged to tell mine.

I came to Austin with the single purpose of doing a quick and painless Masters in Computer Engineering. I didnt view the PhD even as a remotely plausible option, nor did i believe i was capable of doing one. I had corresponded with a couple of professors, including Dr. Jacob Abraham about some future projects. I remember the first time i met Jacob. It was a hot Austin August afternoon, when i had been doing the rounds, looking for a TA. He was running from his office to his class and had stopped to welcome me en route. He spoke about his VLSI course, FPGAs, Patrick Cousot, how he planned that i should do static analysis for my PhD, CERC, group meetings, pizzas, Motorola, RA, and raced over some other topics as well. As he ran at top speed toward his class again, i was struck by the fiery, animated, zeal that i had just witnessed. From that day to this one, a conversation with Jacob always sends me whirring into a zone of drive and enthusiasm. I have never heard him sound disappointed about anything- he always visualizes a positive outcome and veers people towards it. In the second conversation that i had with him, he effusively chartered the road map for my Masters, my PhD research, and the steps toward applying for a faculty position at excellent institutions. While most people would be cautious and circumspect, mapping the future of a brand new student, Jacob aspired freely. And sitting self-conciously in this stalwart's imposing office, i learnt how to dream. I remember the thrilled disbelief i felt that day, associating these stellar achievements with myself. From then on, ever so often, when he

felt positive about my progress, he would dream for me. Every such dream was a shot in the arm and i would work doubly hard to justify his expectations. If something could make Jacob proud, i was willing to work on it. Once, early in my PhD, i had missed a paper deadline and was convinced that i needed to quit the program, when he walked into my cubicle, spoke about the philosophy of research, about perspectives and ambition. He concluded by saying “i know that my star researcher will go a long way”.

A couple of years into my PhD, i had a serious family emergency that i wanted to attend to immediately. I was going to fly home to India in the middle of the semester, with no idea whether i would be back for finishing the degree. I had planned to leave within a day’s notice, and the stress of the situation was stretching me like elastic. I stood outside Jacob’s office, fearing that i would never come back to this world again. When i met him, Jacob’s face was concerned, but he had no doubt that i would come back, finish my PhD and do an excellent job on it. “How do you know?”, i asked, fearing that he didnt. He stared into my face and said “I know i’d be very surprised if you didnt get a faculty position in one of the top 5 universities of the country.” Today, as i sit reminiscing in my office in UIUC, i know i wouldnt have been in these portals, if not for Jacob. Thank you Sir, for that day and those years. Thank you for making me want to try so hard.

To the extent that the work in this thesis deserves credit, a lot of it goes to Dr. Allen Emerson. He has spent considerable time in discussing and thinking through this work with an intensity that only he is capable of. His deep insights into the science of computing, his commitment to purity and his restless quest for mathematical truths have served as a lighthouse of inspiration for me through my PhD. I feel truly fortunate to have had extensive contact with a paragon of academic idealism like Prof Emerson. I remember one of the first times that i discussed an idea with him, when he wrote “Excellent. I wish you were my student” I will always count that moment as a high honour in my academic career. Thank you Prof Emerson, for teaching me the value of perfection and the importance of striving for it.

Dr. Jason Baumgartner has always responded with alacrity and precision to all my queries, and has been a very motivating and encouraging mentor through my graduate career. I thank him for his involvement in my work. I would also like to thank Dr. Aziz and Dr. Garg for their guidance through my PhD.

I truly believe that the friends i made during my stint at Austin constitute the biggest achievements of my life. Every one of those friendships has been as rewarding as a doctorate! Thanking my friends is like measuring the surface area of the earth- i wouldnt know where to start and how to finish. However, here is an honest attempt at the task.

Siddarth Krishnan has bustled, befuddled, bumbled and bamboozled his way into my life at UT Austin. I am not able to recollect a single day or event in Austin without his presence. The winsome wit, the jocular joy, the tireless teasing, the gleaming goodness-all of it blazed like a Titan on every day of my life. We have been through countless systoles and diastoles together; every rung we stepped on, amidst noisy quibbles and banter, took us further on the ladder of growing up. I used to seek his involvement and approval in every act of mine and i admit that is still true. Things just seem a lot more worth doing, if Sid is a part of them. When i think of years with Siddarth, i reminisce about the endless Starbucks jabber jaunts, Einstein bagel indiscretions, Blockbuster visits, rock climbing expeditions, culinary concertos, vigorous car music...but most of all i remember the Perennial Prattle and Continuous Chatter. I will always remember Siddarth with deep gratitude and fondness for making my darkest days in Austin seem bearable. I was going through a personal crisis and my future seemed tenuous. Siddarth took over the task of propping me up and buoying my spirits, with a pilot-like responsibility. Overnight, he had transformed from flippant fun-monger to a pillar of positivity. When he left Austin, i never really came to terms with his leaving, and always felt like i was running on a spare tyre. As i hobbled through the remaining days of my PhD, i felt i was tiptoeing on the periphery of the yawning chasm he had left

behind. Thank you Siddarth, for having the gift of joy, and gifting me with it.

Vinod Viswanath has a place in virutally every part of this thesis. Starting from my first class at UT, he has been there throughout- steady, strong, supportive, sympathetic, synergetic and sprightly. I think i can state without exaggeration that Vinod has been seminal to my career and life as a graduate student. Ever since i discovered his ability to galvanize people and root for them, i would traject his boundless enthusiasm toward my newest idea. I have interacted incessantly with him on discussing ideas, writing papers, making presentations, classes, exams, qualifiers...a lion's share of this work would not have been completed, but for him. But Vinod's biggest contribution to my PhD is not his collaboration on the work itself. Vinod taught me about VLSI for vocation and California for vacation; he drove me to my goals and taught me to drive; he taught me how to use a map, and how never to never use a mop; the indulgence of Krispy Kreme and the abstinence from harsh thought and words; the thrills of Vegas and the trill of the Vedas; the importance of being earnest and of earning your importance; the opulence of Sanskrit poetry and the paucity of spartan living; Vinod has taught me most of what he knew and what i know. My warmest memories of Vinod in Austin are of labyrinthine drives, of zillionth sitcom reruns, of Ken's donuts on hungry all-nighters, of JP's Yavadvipa, of fussy festival days, of quaint melodies, of strawberries and bananas, of rainy Austin days. During the time that i was looking for a job, Vinod helped me through every waking day. The applications, the strategies, the interview preparations, will all stay lush in my consciousness for as far as my career will take me. Thank you Vinod, for being. And for letting me be.

Sumant Kowshik emerged in my life one fine day, and soon after, started to belong there. We had a strange friendship- one where we rarely met each other, but spoke ceaselessly on phone. What started as a voice from out of the bluetooth, stood steadfast as the voice of reason and moderation. Day after day, call after call, he would endure, soothe, caution, advice, worry, fret and laugh. Like a banyan

tree in a tornado, he would waver slightly for every inconsistency of mine, only to turn up the next phonecall. He never talked about himself, but made my concerns his business. Whether it was booking air tickets, assuaging my frequent fears or making me laugh, he anchored the “shipwreck” back to the safe waters with each act of his. When i was in India for an extended, bleak period of my life, he was my sole contact with the world i had left behind. The nightly phonecalls he made, were a draft of air on muggy, throttling days. He presented the arguments for resuming my PhD, and in his thorough, cogent manner, convinced me to come back. In the theatrical production of my PhD, Sumant’s curtain call would get a standing ovation. My most affectionate memories of Sumant include chatanddebateandgossipandgabandbabbleandrantandcantandchitchatandargueandtalkandtalk- poetry, new New Yorkers, bollywood, India Inc., Real Shankaracharya, making an advertisement, Chicago, Phoenix, Rocky mountains, the business of being funny, funny businesses, family, flaws, fine food and filibuster forever. We have had a conversation that has lasted for years. Thank you Sumant, for your understanding, of me and the world. I believe in messiahs now.

Kunal was a source of reassurance and energy during my PhD. In the 10 years that i have known him, convlusive laughter, quick quips and positive reinforcement have been the theme of our friendship. In the past few years, my best memories of him are on fun packed vacations, programming assignments, Thai noodles, clueless cluenetics, querulous quizzes and unquestioning rides to school. Mulla and Ramdas were centric to much mirth, music and rejoicing. Bindhu proved to be an invaluable asset during the days before my defense. She promised me a house and gave me a home.

I am very thankful to the support provided by Debi, Melissa, Linda Frost, Shirley and Andrew. I am particularly thankful to Melanie Gulick for the empathy that makes her go the extra mile for students. I want to thank Ruth Ann Abraham for the sensitivity and warmth that she has shown me and my parents repeatedly.

I owe everything i am about to my family. From my father, i have learnt the lessons of ambition, discipline, perseverance and success. From my mother, i have learnt about beauty, tradition, strength, compassion and happiness. From my sister, about passion, action, emotion and grit. Together, these people have given me life and and taught me how to live it. My parents imparted their spirit to me, but stepped back enough to let me seek and find my place the world. Now that i have found it, i want to thank them for unfastening the clasp early in my life, and letting me soar. Today, when i am one of a meagre minority of women faculty in a top engineering program of the world, and there is so much talk of mistaken perceptions that dont let women do well for themselves, i know what my secret is. I know that from the day i was born, my parents expected me to do as well as did their friends who had sons. They never bought me dolls, never worried about me cooking or keeping house, encouraged me to assert myself, urged me to perform on stage, taught me to stand up to injustice; they didnt follow a single societal stereotype while parenting a female child. That is why, when i see unfortunate perceptions that hinder womens' progress, i thank my family a million times. Thank you Amma, Daddy and Vandana, for making me who i am.

High Level Static Analysis of System Descriptions for Taming Verification Complexity

Publication No. _____

Shobha Vasudevan, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Jacob A. Abraham

The growing complexity of VLSI and System-on-a-chip(SoC) designs has made their verification extremely expensive, time-consuming and resource intensive. Formal verification of system behavior is critical to the design cycle due to its ability to isolate subtle flaws and provide high quality assurance. However, its computational intractability limits applicability in practice, rendering the state-of-the-art insufficient to meet the needs of the industry.

In this dissertation, a suite of techniques that are a significant departure from traditional Boolean level approaches to formal verification are presented. The algorithms are based on a top-down, domain-aware perspective of the system by reasoning at the system level and register transfer level (RTL) descriptions. Static analysis of these high level system descriptions using structural information and symbolic reasoning leads to effective decomposition strategies that create tractable portions of the system. These manageable system components can then be verified by deploying efficient Boolean level algorithms. The techniques presented here apply to the actual RTL source code, and are intended to blend seamlessly into the system design cycle.

All approaches using high level static analyis follow a three pronged solution- domain aware analysis, high level symbolic simulation and a decision procedure for the verification task. This work advocates a marked difference in the perspective to formal hardware verification as compared to popular paradigms. The techniques shown here are illustrative of a hardware-aware viewpoint to verification, and argue this case for contemporary verification tasks.

Antecedent conditioned slicing, an abstraction technique for reducing RTL design space is introduced. The reduced RTL can then be model checked. An open source RTL implementation of the USB 2.0 protocol is verified using this technique.

A technique for pipelined processor verification using antecedent conditioned slicing is introduced. An open source Verilog RTL of OR1200, an embedded processor is explained in a detailed case study for verification using this technique.

A static analysis technique is proposed to alleviate the complexity of the sequential equivalence checking problem between system level and RTL descriptions, by efficiently decomposing designs using *sequential compare points*. A satisfiability (SAT) solver is used as the lower level verification engine. In a case study, sequential equivalence checking between a system level description of a Viterbi decoder and two different RTL implementations are detailed.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Formal methods is the name given to a group of analytical techniques that provide a framework of mathematical and logical reasoning. Formal methods have been effectively employed at various stages in the process of building computer systems. The expectation while using formal methods is that the rigor of mathematical analyses during the specification, development and verification of systems will lead to their robustness and reliability.

In particular, the paramount merit of using formal methods in the verification of systems is universally agreed upon for many reasons. Correctness is the fundamental concern of the theory of computer science or its application to engineering software, hardware or embedded systems. Although program verification by proof is an absolute scientific ideal that can be pursued for its own sake, its practical ramifications span from saving life and property to commercial and economic benefits. A known flaw can cost billions of dollars to a commercial hardware or embedded systems enterprise, since unlike software, versions of these systems cannot be corrected and released, once they are fabricated. Errors in the design detected before fabrication can also be detrimental to development schedules, as well as expensive to correct. The primary focus of this thesis will be around these digital hardware and embedded systems.

Despite the critical nature of the issue of building reliable systems, the advances in incorporating foolproof verification practices have been found wanting. At present, the most popular means of increasing confidence in systems is by simulation based verification or testing, which is the application

of random or predetermined input patterns to check the behavior of the system. Testing is often ad-hoc and always incomplete. Exhaustive simulation of any reasonably sized system is practically impossible. The effectiveness of a testing strategy depends on the sufficiency of the coverage metrics, which are ill-characterized. Even when the coverage metrics are defined, creating a set of input test patterns that can appropriately target the coverage metrics is a very difficult task. Consequently, with the growing complexity of digital systems, a disproportionate amount of money and time is spent on testing. This often results in the testing teams having more strength than the development teams and three-fourths of the development cycle being spent in testing. The growing complexity and size of the digital systems makes their validation a daunting challenge. Digital systems have grown to mammoth scales, with over ten million transistors integrated on a single chip. This breakthrough in technology has, in fact, reached the point, where it is hard to design a complete system from scratch. With the advent of System-on-chip (SoC) designs that usually involve the integration of heterogeneous components on a single circuit, unprecedented levels of complexity and size have been attained. Consequently, the validation of such systems is one of the greatest bottlenecks in their development.

1.1 Formal Verification Overview

Formal verification emerged a few decades ago as a viable candidate solution for the validation crisis, due to its ability to isolate subtle flaws and provide high quality assurance. Formal verification involves mathematical proofs of the consistency of an implementation with a rigorous formulation of selected key requirements, often called the specification.

A discussion on formal verification, however, is always tailgated by a reference to its complexity. Due to the exhaustive nature of the formal verification task, it is computationally intractable for anything more

than moderately sized systems. Formal verification can be classified broadly into two categories – interactive deductive verification and automated finite state machine verification based on state enumeration.

State-space based techniques like model checking[58], BDD-based verification [34, 41] etc., reason with the state space of the entire design. Deductive verification techniques like theorem proving [144, 207], rewriting [142] etc., try to solve the verification problem by equational reasoning. The computational intractability of the verification manifests as *state-space explosion* in automatic state-space based methods leading to practical time and space limitations. In the case of deductive methods that circumvent state space explosion and are size independent, the computational complexity manifests itself as a lower degree of automation, requiring manual intervention during the verification process. Therefore, while state-space based approaches cannot handle circuits of even reasonable sizes, deductive verification approaches involve a significant manual component.

1.2 Hardware Design Cycle

In order to appreciate the status and role of formal verification in hardware, it is important to get acquainted with the rungs in the design ladder. At the highest level of integrated circuit design is the specification, or the statement of intent, usually drafted by system architects. This is usually a document written in English. This document often suffices as a specification, although in the SoC industry, specification models or simulators are on the increase. These are described at the Electronic System Level (ESL) such that the behavior of the entire system is captured with more precision than in English text, to enable accuracy and ease of implementation. ESL languages include C/C++ and their derivatives like System C and Spec C.

The next step of abstraction can also be viewed as the first stage of implementation of the design. The implementation in the case of hardware is not the actual physical realization of the hardware, but a more

abstract, technology independent model, called the *design*. This is the Register Transfer Level (RTL) which describes the operation of a synchronous digital circuit in terms of data transfer between registers or flip-flops. Circuit synchronization is done by the registers with respect to an explicit clock, while combinational logic performs the functions of the circuit between any two registers. At this level, the logical and timing accuracy of the circuit is described, using a Hardware Description Language (HDL), that is usually one of VHDL or Verilog. Although the combinational logic represents logical functions, it does not explicitly encode the circuit in terms of Boolean gates. The RTL design, therefore, is an abstract model constructed before spending time and resources on the physical realization of the design. Since the faithfulness of the RTL design with respect to the specifications is critical to further stages of development, all verification is performed at this level. Simulation based verification is the principal task that is performed during the RTL design phase. Formal verification is also applied at this level. After multiple iterations between the verification and design teams, the RTL is considered stable. Formal verification, then, ensures the functional accuracy of the design, but does not provide any guarantees about the timing and physical factors in the circuit.

Logic synthesis maps the RTL design into the fundamental Boolean logic gates with an explicit wiring of the circuit. The gate level netlist is a technology aware phase of the design process. The gates are placed and routed according to strategic optimizations dependent on power, area and timing constraints. The clock is physically realized as a balanced tree at the netlist level. The layout of the entire chip is created from the logic gates with the corresponding transistor devices. This is then fabricated and taped out.

1.3 Practical Realities with Applying Formal Verification

Hardware designs have been the source of the biggest success stories of formal verification methods, as compared to other domains like software programs, protocols etc. This is primarily due to the unruly number of reachable states in other domains as compared to digital hardware whose range is limited to Boolean values. Triumphs of formal verification in hardware include pipelined processors[42], superscalar processors[172, 208, 241, 242], floating point units[144], cache coherence protocols and many more that havent been enumerated. However, the application of these techniques in a contemporary industrial environment is extremely limited. Formal verification comprises a minor portion of the resources and techniques of a typical hardware industry validation project. In fact, there are many large scale validation efforts in mid-sized (or large) semiconductor ventures that are devoid of any formal verification methods. Formal verification is perceived as conceptually desirable, but impracticable by a large number of technocrats in industry. Among the two streams of formal verification, deductive techniques have sometimes been employed in an industrial context [75, 169]. The factor that limits their widespread applicability is the user expertise and effort that accompanies them. In the case of automatic techniques, model checking has been applied in various cases [133] to varying degrees of success. Typically, this comprises an isolated functional unit of the design, like a floating point unit or a protocol implementation, where the chances of subtle flaws are high. Since model checking and its derivatives frequently runs into capacity issues, the complete power of model checking is not exploited by the existing verification environment. At present, formal verification at the full chip level remains a distant dream.

The electronic design automation (EDA) industry has consistently invented generic, state space based algorithms that operate at the Boolean gates level. Model checking based techniques like Symbolic Trajectory Evaluation, STE [36] counterexample guided refinement [61], proof based abstraction, as

well as Boolean satisfiability (SAT) based verification [213] and Automatic Test Pattern Generation (ATPG) based verification reason with Boolean gates. Although most of them can accept RTL as input, the model checking algorithms are applied to the gate level state space of the circuit, by synthesizing the RTL into an internal gate level representation. Even in the case of more sophisticated algorithms, where the internal representation may not explicitly enumerate all the states, the application of the technique is entirely with respect to Boolean state encodings. In other words, state of the art tools and techniques reason with the post-encoding artifact, instead of the pre-encoding one.

The pre-encoding entity, or RTL was developed based on the premise that it would be easier to verify a more abstract model that outlined desired functionality, as opposed to a very detailed one. This is the reason why simulation based verification in all industrial environments are carried at the RTL. Why, then, are formal verification techniques applied at the gate level, thereby losing the advantage of RTL? The answer to this question is probably more due to historic reasons, as opposed to concious strategy on the part of CAD tool developers.

A Hardware Description Language (HDL) is a standard text-based expression of the temporal behaviour and/or (spatial) circuit structure of an electronic system. In contrast to a software programming language, an HDL's syntax and semantics include explicit notations for expressing time and concurrency which are the primary attributes of hardware. Historically, VHDL and Verilog, both HDLs used to describe RTL were initiated by the US Department of Defense and Gateway Design Automation independently.

The introduction of logic-synthesis for HDLs pushed HDLs from the background into the foreground of digital-design. Synthesis tools compiled HDL-source files into a manufacturable gate/transistor-level netlist description. Writing synthesizeable RTL files required practice and discipline on the part of the designer; compared to a traditional schematic-layout, synthesized-RTL netlists were almost always

larger in area and slower in performance. Circuit design by a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always outperform its logically-synthesized equivalent, but synthesis's productivity advantage soon displaced digital schematic-capture to exactly those areas which were problematic for RTL-synthesis: extremely high-speed, low-power, or asynchronous circuitry. In short, logic synthesis not only propelled HDLs into a central role for digital design, it was a revolutionary technology for the system design industry.

The IEEE standardization for VHDL was done in 1987, and for Verilog in 2001.

Despite their growing popularity, these languages were widely perceived as non-formal and their computational models were not well defined. In fact, the simulation semantics of Verilog and VHDL are similar but not identical, due to which there are inconsistencies in the description of RTL between these two languages. Due to these reasons, possibly, the formal verification community was more comfortable applying their algorithms to the well defined gate level models. Subsequently, with their standardization as well as popularity, the synthesizable subsets of the HDLs is quite well understood and accepted. In the current scenario, when despite massive CAD tool enterprises, the efficacy of formal verification is limited, a revision in thinking is probably in order.

1.4 Philosophy of the Dissertation

The principal philosophy behind the work in this dissertation is that hardware verification algorithms and tools need to make a paradigm shift in order to be able to scale to the growing demands of the system design industry. This paradigm shift, if made "upwards" in the design cycle, would be able to scale to larger and more complex designs. In other words, formal verification should be employed at the source code levels (RTL and higher) in order to take meaningful steps toward scalability.

State of the art CAD algorithms in the EDA industry perform better than their predecessors on benchmark designs. Depending on the number of benchmarks, this is an excellent testimony to their genericity. However, they are often rendered ineffective when applied to even a moderately sized design. This approach of building generic tools that are supposed to perform equally on any given design has not yielded as much practical impact as desired. Since formal verification strategies reason with the Boolean state transition graph, they are unable to exploit any information at that level. These algorithms are not cognizant of the design they are checking. Due to the Boolean quantification, state transition relations at the gate level are extremely detailed and inscrutable, making it impossible to distinguish features of one design from another. The gate level tools and techniques, therefore, are unaware of the domain they are working in, and are not able to manipulate any information from the domain.

This thesis, then, proposes and argues for a different viewpoint than the traditional perception of formal hardware verification techniques and tools. The primary intent of this work is to demonstrate and thereby motivate the benefit of high level reasoning by statically analyzing source code, in conjunction with existing tools and algorithms, to form complete formal verification solutions for large design spaces.The generic name given to the comprehensive set of techniques, perspective and philosophy, hereafter is *high level static analysis*.

Since RTL of a design is developed by a human, it is modular, informative and clearly distinct from other RTL designs. It is conceivable, then to select a verification strategy according to the type of design. Since RTL is usually reused from legacy designs from generation to generation, and since there has been relative stabilization in RTL design practices, the RTL descriptions for broad classes of circuits are typically very similar. This similarity usually exists across a very broad domain of circuits-for instance RTL descriptions of pipelined processors follow a very similar syntactic structure, as do arithmetic circuits. If the operational semantics of a design is similar to another, there is a high probability of a

similarity in structure. Therefore, it is possible to devise techniques that are applicable to the domain of circuits with a similar operation (and structure) if we reason with the RTL design. The domain aware analysis would decompose the design into smaller, more tractable components that can be verified. This would probably sacrifice the genericity of the algorithm, but as shown in the rest of this dissertation, would be able to solve very challenging formal verification problems in a given domain. Since this domain awareness is more "hardware oriented" and is closer to the way a hardware design engineer would solve a problem, it might be the best way to increase designer productivity and bridge the gap between the formal verification teams and the design teams.

Reasoning at higher levels of abstraction is always accompanied by an increase in efficiency and scale [135]. The RTL design descriptions encapsulate the gate level Boolean entities into more abstract representations. The RTL operators do not operate on one bit at a time, but on entire bit vectors. Instead of explicit state transition graphs, control flow graphs of the RTL are manipulated. These are necessarily smaller representations. This is due to the grouping of sets of states and transitions.

Data can be abstracted by leaving functions uninterpreted. Equational reasoning can be applied when appropriate, over the RTL function symbols. Syntactic structure of the HDL code can be exploited for using abstraction mechanisms similar to software. It might not be useful to reason entirely with RTL in some situations. In those cases, the pre-processing or initial high level analysis would be an excellent support to existing sophisticated verification tools. The top-down approach will be able to identify the lower level engines that are most suited to the verification task. In summary, the emphasis of this way of thinking is in flexibility of the formal verification approach, with the goal of verifying a given design. Many of the RTL analysis techniques explored in this dissertation are applied synergistically with lower level automatic engines, thereby bolstering the use of these well researched algorithms.

1.5 Approach of the Dissertation: High Level Static Analysis

In this work, high level static analysis techniques for verification are explored. Although the individual techniques depend on the target application domain, all of them adhere to the principle of easing verification tasks that are found challenging by existing engines. Since the practical applicability of verification is of cardinal significance to this work, each static analysis based verification technique is explained with respect to a non-trivial case study. The focus of each piece of work, then, is the target hardware design domain, around which an appropriate static analysis technique is developed. Every contribution presented in the paper follows a three-pronged solution.

- Domain aware analysis
- High level symbolic simulation
- Decision procedure

Domain aware analysis

In keeping with the philosophy of this top-down approach to formal verification, the static analysis is aware of the domain of the target application. This means that algorithms and heuristics that best fit the RTL description are employed. These algorithms are used for decomposing the monolithic verification problem into smaller components. The smaller components are verified using a higher or lower level engine. The domain awareness also provides a perspective on the lower level engines that can be synergistically used with the static analysis technique. The domain awareness of the solution will be evident in every case study that is a part of this dissertation.

High level symbolic simulation

This is the mainstay of all the static analysis techniques proposed in this work. In order to analyze entities

in RTL, it is desirable to simulate their behavior. While logic simulation manipulates scalars, symbolic simulation builds expressions, or formulas, instead of the scalar values in a logic simulator. Symbolic simulation computes expressions at a circuit output in terms of logical functions of its inputs. Bit-level symbolic simulation [83] is applied strictly at the gate level of a circuit using Binary Decision Diagrams (BDDs). A single run on a bit level symbolic simulation captures many runs of a scalar simulator.

Since high level static analysis seeks to analyze the RTL source code at the pre-encoding stage, the techniques in this work do not use bit-level symbolic simulation. The structural information of the design is available from the RTL source code. This is expressed in the syntax of the Hardware Description Language (HDL). The semantics of the source code, or the actual behavior of the design can be obtained by a different representation of symbolic simulation, called high level symbolic simulation. High level symbolic simulation [135] has been defined and applied to processor verification successfully. All the techniques described in this work uniformly employ high level symbolic simulation during the analysis or the verification phase. High level symbolic simulation builds expressions with algebraic terms, based on a subset of quantifier free first order logic. This approach is at a much higher level of abstraction than the gate level and is discussed later in detail.

Decision procedure

High level symbolic simulation operates within the framework of a quantifier free subset of first order logic with uninterpreted functions [42]. A function symbol could be left uninterpreted in this process, in order to provide data abstraction. The verification engine would now provide a Boolean interpretation to these functions. High level symbolic simulation could be the basis for the verification algorithm itself or it could be used as a part of the static analysis that helps decompose the design state space. With the latter, different existing Boolean algorithm would need to be applied to provide a Boolean interpretation to the symbolically simulated expressions. The choice of this “interpreting procedure” is dependent on

the application and the verification task at hand. A high level rewriter could be more effective than a SAT solver in a case where the high level symbolic simulation does not yield a decomposition that is tractable by the SAT solver. The decision procedure in every technique is therefore contingent on a number of factors, and is an important part of the high level static analysis. It is of considerable value to existing Boolean level checkers to know which algorithm to deploy, so as to save time and resources without trial and error.

1.6 Contributions

The intent of this dissertation is to demonstrate the benefit of using high level static analysis of system descriptions. High level symbolic simulation is the foundation of all the techniques presented here. Domain aware analysis is also a departure from traditional formal hardware verification approaches. We define high level symbolic simulation for RTL and provide a reasoning for the efficiency of high level analysis over lower level analysis. To the best of our knowledge, this methodology has not been presented before. This mode of thinking is the essence and primary contribution of the dissertation. In order to provide a persuasive argument for this rationale, we present algorithms that use our three pronged approach to high level static analysis, and apply them to non-trivial case studies.

The three main contributions of the work are listed below.

1.6.1 Sequential Equivalence Checking between ESL and RTL

Sequential equivalence checking between system level descriptions of designs and their Register Transfer Level(RTL) implementations is a very challenging and important problem in the context of Systems on a Chip (SoCs). We propose a technique to alleviate the complexity of the equivalence checking problem, by efficiently decomposing it using compare points. Traditionally, equivalence checking

techniques use nominal or functional mapping of latches as compare points. Since we operate at a level where design descriptions are in System Level Languages or Hardware Description Languages, we leverage the information available to us at this level in deducing *sequential compare points*. Sequential compare points encapsulate the sequential behavior of designs and are obtained by statically analyzing the design descriptions. We decompose the design using sequential compare points and represent the design behavior at these compare points by symbolic expressions. We use a SAT solver to check the equivalence of the symbolic expressions. In order to demonstrate our technique, we present results on a non-trivial case study. We show an equivalence check between a System C description and two different Verilog RTL implementations of a Viterbi decoder, that is a component of the DRM SoC.subsectionAntecedent Conditioned Slicing

1.6.2 Antecedent Conditioned Slicing

Static slicing has shown itself to be a valuable tool, facilitating the verification of hardware designs. In this paper, we present a sharpened notion, *antecedent conditioned slicing* that provides a more effective abstraction for reducing the size of the state space. In antecedent conditioned slicing, extra information from the antecedent is used to permit greater pruning of the state space. In a previous version of this paper, we applied antecedent conditioned slicing to safety properties of the form $G(antecedent \Rightarrow consequent)$ where $antecedent$ and $consequent$ were written in propositional logic. In this paper, we use antecedent conditioned slicing to handle safety and bounded liveness property specifications written in linear time temporal logic (LTL). We present a theoretical justification of our technique. We provide experimental results on a Verilog RTL implementation of the USB 2.0 functional core, which is a large design with about 1100 state elements (10^{331} states). The results demonstrate that the technique provides significant performance benefits over static program slicing using state-of-the-art model checkers.

1.6.3 Processor Verification Using Antecedent Conditioned Slicing

We present a technique for automatic verification of pipelined microprocessors using model checking. *Antecedent conditioned slicing* is an efficient abstraction technique for hardware designs at the Register Transfer Level (RTL). Antecedent conditioned slicing prunes the verification state space, using information from the antecedent of a given LTL property. In this work, we model instructions of a pipelined processor into LTL properties, such that the instruction opcode forms the antecedent. We use antecedent conditioned slicing to decompose the problem space of pipelined processor verification on an instruction by instruction basis. We discharge the resulting smaller, tractable problems using an automated verification engine.

We thereby verify that every instruction behaves according to the specification, and ensure that non-target registers are not modified by the instruction. We apply this technique to verify all instruction classes of a Verilog RTL implementation of the OR1200, an off-the-shelf pipelined processor. We study the impact of antecedent conditioned slicing to facilitate a variety of verification engines, including BDD-based reachability, interpolation, abstraction-refinement, STE, and induction.

1.7 Outline of Dissertation

The outline of this dissertation is as follows. Chapter 2 explains the important concepts with respect to system level languages and RTL that are required to establish the relevant context and background. A technique for combinational equivalence checking at RTL is presented in Chapter 3. This technique details a solution using the three pronged static analysis approach and also demonstrates results on arithmetic circuits. Chapter 4 extends the notion of equivalence checking to sequential designs, and presents a high level static analysis technique for sequential equivalence checking between system level

specifications and their RTL implementations. In Chapter 5, an RTL abstraction technique is introduced, and its in reducing the state space of model checking is shown. In Chapter 6, the same technique is applied to processor verification, and through top down domain aware analysis, is shown to be an apposite fit for this task. A notion of correctness of pipelined processors is provided with this approach. The thesis concludes with Chapter 7.

Chapter 2

Understanding High Level System Descriptions

The trend in the hardware and the SoC industry is to ascend in the level of abstraction for description of design behavior. The collective shift from gate level descriptions to RTL about a decade ago is now promising to be replaced by a shift toward system level languages. The compelling need for higher levels of abstraction is due to the bottlenecks that validation and verification of SoCs and digital hardware pose. As RTL modeling sped up design simulation over gate level, system level languages afford the same benefit over RTL. Additionally, the entire system can be validated (by simulation) using the executable platform used to develop the software. System level languages include System C, Spec C, Handel C etc., although a single popular candidate is yet to emerge. The abstraction of designs using these languages is typically modeled at transaction level , where the details of communication among computation components are separated from the details of computation components. Communication is modeled by channels, while transaction requests take place by calling interface functions of these channel models. The other commonly occurring model is behavior level. In general, behavioral level code is written using the same data types as RTL, but it is unscheduled and unallocated. That is, the data objects and the operations on them are specified, but the resources and the schedule of cycles required are not specified.

Describing designs at a higher level of abstraction also helps identify the conformance of power, area and timing to their respective budgets, during the early design analysis phase.

It is therefore highly germane and futuristic to reason with these high level system descriptions. In order to apply formal verification at this level, an emphasis should be placed on the semantics of the language beyond simulation alone. Some work on formalizing the semantics of Verilog and VHDL has been done [101]. This chapter attempts to grasp the formal ramifications of high level reasoning.

2.1 Hardware Description Languages

Hardware Description Languages (HDLs) are used to describe circuit designs at the behavioral and the register transfer level. These languages are seemingly similar to software, but differ in their ability to depict timing and concurrency, pertinent to hardware. Hardware systems are concurrent in the complete sense of the word, since every logic gate in the system simultaneously evaluates its output as a function of its inputs. Therefore, they receive inputs and outputs continuously from their environment without halting. Verilog and VHDL are the most popular HDLs in the semiconductor landscape. Although structurally similar to C and ADA respectively, these languages do not model recursion or pointer arithmetic, that are irrelevant to hardware. HDLs are used to write executable specifications of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is physically realized. The HDL program is then "compiled", which in this case means synthesis, a process of transforming the HDL code-listing into a gate level netlist.

2.1.1 Synthesizable subset

Among the constructs of Verilog, a subset has been identified as the one that can be realized into hardware, and can be called the synthesizable subset of Verilog. The primary features of this synthesizable

subset are that it avoids unbalanced structures, ill-defined or multiply defined variables, explicit timing definitions, and other constructs that have an ambiguous hardware correspondence.

A synchronous circuit consists of two kinds of elements: registers and combinational logic. Register synchronize the circuit's operation to the edges of an explicit clock signal, and are the only elements in the circuit that have memory properties. Registers are modeled as flip-flops. Combinational logic performs all the logical functions in the circuit and it typically consists of logic gates. When designing digital integrated circuits with a hardware description language, the designs are usually engineered at a higher level of abstraction than transistor or gate level. In HDLs the designer declares the registers that roughly corresponds to variables in computer programming languages, and describes the combination logic by using constructs that are familiar from programming languages such as if-then-else and arithmetic operations. This level is called register transfer level. The term refers to the fact that RTL focuses on describing the flow of signals between registers.

A Verilog design consists of a hierarchy of modules. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behaviour of the module by defining the relationships between the ports, wires, and registers. Statements that are executed in a sequence are placed inside an `always` block and executed in sequential order within the block. All concurrent statements and all `always` blocks in the design are executed in parallel. A module can also contain one or more instances of another module to define sub-behavior. It is assumed that there is a single clock input to the program whose single cycle comprises two logic levels, flagged by a positive and a negative edge. The environment is such that no other inputs change when the positive edge of the clock arrives. A flip-flop is a single register with an input and output that is clocked on the positive edge of the clock. An HDL representation of this is a program that can be partitioned into a set of concurrently executing entities (threads) that are

a combinational program and a set of flip-flops. It is also assumed that no two distinct flip-flops drive the same output, to avoid race conditions. A simulation cycle starts with a positive edge of clock. The positive edge enables all the flip flops and none of the other threads. When all the output variables are updated, the simulation cycle quiesces or converges. At the negative edge of clock, none of the flip flops get enabled. The combinational threads, however, converge with each variable getting updated with the value of the expression that updates it.

When the simulation cycle quiesces or converges to a sequence of states at successive instants of clock, the simulation model is said to follow trace semantics. This is conducive to formal verification. Most practical formal methods are oriented towards system descriptions in terms of sequences of states, or the trace semantics. In contrast, event-based changes to the values of registers is called event semantics of HDLs. Event semantics can accurately model asynchronous behavior, but is not amenable to formal verification. RTL designs are always implemented to display synchronous behavior, which is why the trace semantics should be followed when simulating an RTL. In other words, asynchronous circuits are not modeled in RTL, therefore their behavior is out of the scope of this thesis.

The principal synthesizable Verilog RTL constructs are detailed in order to assist understand the syntax and semantics of the code fragments used often as examples in the rest of the thesis.

Modules

The module is a fundamental building block of Verilog. A module can be viewed as a self-contained functional block in the hardware. It consists of input and output ports, wires and registers. While synthesis, there is no gate level analog of the module, since the hierarchy is flattened. However, the modularalization is intuitive and facilitates the hierarchical construction of the design. A module usually contain more than one concurrently executing block. A module can be instantiated with different

parameters when it is a sub-component of another module.

`always` blocks

An `always` block depicts a concurrently executing process in Verilog. It consists of a set of statements and a list of signals that can trigger its execution, called the sensitivity list. In a synchronous transition system, this list will include the clock. When a signal present in the sensitivity list changes its logic value, all the statements in the `always` block get executed. The logic synthesizer ignores asynchronous triggers in the sensitivity list. An `always` block has an implicit infinite loop, causing it to run forever.

Procedural statements

Instead of describing a component using primitive Boolean gates, RTL allows expressions to be assigned to describe the logic. Verilog `assign` statements can be used for this purpose, providing a functional abstraction over the explicit Boolean gates. Two different types of procedural assignments are allowed in Verilog-blocking and non-blocking. Blocking assignments block the execution flow into the procedure until the assignment is completed and are denoted by `=`. Non-blocking assignments evaluate everything on the right hand side, but postpone the transfer of values to variables on the left hand side until all evaluations in the current time step are completed.

Other high level procedural statements include `if-then-else` to describe a 2:1 multiplexer, `case` statements and bounded `for` and `while` loops.

Flip-flops

The RTL representation of an actual piece of hardware memory element is the flip-flop. A flip-flop can be modeled within an `always` block as a variable sensitive to the input clock signal, with non-blocking assignments to that variable.

Vectors

A register or wire that have more than single bit widths, can be expressed as bit vectors or arrays in Verilog. Bitwise addressing is possible within these arrays.

Operators

Along with Boolean operators, relational and arithmetic operators are also allowed in Verilog. These operators can operate on the individual bits in a variable, as well as bit vectors. Arithmetic operations like $+, -, *, /$ in RTL represent the actual hardware realizations of an adder, a subtractor, a multiplier or a divider. Similarly, the $<<, >>, << n, >> n$ operators denote a one-bit left shifter, a one-bit right shifter, as well as shifters by n places.

2.2 Register Transfer Level Abstractions

An RTL description is usually well partitioned into datapath and control. Datapaths consist of the data components like registers, interconnecting wires and buses etc. as well as the nexus between them. The control portion of the design is usually modeled as interacting state machines. These maintain the "state" of the design, or the quiesced values of all the registers at any clock cycle, and make decisions regarding the next state of the design depending on the current state and input signals.

As noted in the previous section, design description in RTL allows for many abstractions that hide the details of the gate level hardware. This abstract framework, however, has not been hand crafted for the purpose of formal verification, as in the case of many previous high level reasoning techniques. The abstraction is the RTL itself, which is a seminal part of the design flow in any system design environment. In comparison with the hand crafted models used for high level verification in the past, the RTL abstraction is much more detailed, due to its relative "closeness" to hardware, and its subscription to a

synthesizable format.

Verification at RTL, therefore, is uniquely positioned in the landscape of verification techniques. Although it allows for considerable information hiding as compared to gate level, thereby providing significant performance benefits, it is sufficiently close to implementation details. Many formal verification techniques have not been able to enjoy widespread use in the industry, since the custom crafted, high level abstractions they work with, are not perceived as realistic.

2.2.1 Rationale for RTL analysis

The primary benefit in reasoning with RTL is the ability to manipulate expressions, instead of primary gates. We will distinguish the RTL expressions by referring to them as term level expressions, as compared to bit-level expressions. This creates new possibilities for symbolic expression based algorithms. The representation of bit vectors and arrays in place of a bitwise, gate-level notation provides a useful high level manipulation. The primary power of RTL analysis comes from the functional or operator abstractions. These operators can have arrays and bit vectors as operands, as opposed to the corresponding explicit bitwise operations at the gate level. Also, some of the operators are arithmetic and relational, and therefore, much smaller than the gate level hardware realization of these functions.

All these attributes can make a significant difference in the size of the data structure representations in large systems with wide datapaths and complicated operations, as well as in data-dependent control logic.

An intuitive way to compare RTL and gate level of a design, is to compare the data structures that correspond to these descriptions. RTL designs, due to their close structural correspondence with software programs, can easily be expressed as control flow graphs, or their derivatives. In contrast, gate level

designs can be expressed as a Kripke structure, or a state-transition graph, where the bitwise values of each variable is captured into a state, and the bitwise change in value is explicitly modeled by a transition. Since the RTL is synthesized into the gate level netlist, it is implied that the behavior of the RTL should be identical to the netlist. In that case, the RTL control flow graph must correspond to the gate level state transition graph. It can be argued, then, that an "RTL state" or a node in the control flow graph corresponds to an aggregate of gate level states, and an "RTL transition" between control flow graph nodes corresponds to multiple transitions of the state graph.

Since the abstractions in RTL are primarily function and operators on data, the aggregation of states as well as transitions in RTL is along the breadth of the gate level state transition graph. The diameter of the state transition graph will not be different from the control flow graph, since there are no control abstractions in RTL.

2.3 Logical framework of RTL

The quantifier free subset of first order logic of equality with uninterpreted functions [42] has been widely used in the context of high level verification of hardware [39]. In relevant prior art, high level symbolic reasoning of expressions has been defined with respect to this logic [135]. In previous work, to the best of our knowledge, high level models have been defined for the specific purpose of verification, as opposed to working with an RTL model. As such, mapping of RTL constructs into a mathematical logic for the purpose of symbolic reasoning has not yet been explored. A treatment of this mapping has been given here.

$$
\begin{aligned}
expr &::= const \\
&\quad | \; functionsymbol(expr, expr....expr) \\
&\quad | \; ite(expr, expr, expr) \\
&\quad | \; expr = expr \\
const &::= false \\
&\quad | \; true \\
&\quad | \; @symbol
\end{aligned}
$$

Figure 2.1. UEF Logic

2.3.1 Quantifier free logic of equality with uninterpreted functions

The quantifier free logic of equality with uninterpreted functions is a decidable fragment of first order logic. This logic is more expressive than propositional logic, but less expressive than first order logic. The abstract syntax is illustrated below. The `ite` operator represents the if-then-else structure, that can be used in conjunction with `true`and `false` to represent all Boolean operators. Formulas in this logic can contain functions that need not have any particular meaning. These functions are called uninterpreted functions. Validity checking in this logic is the process of checking if a formula in this logic holds true for all possible numerical evaluations in its domain. Due to the higher expressive power of this logic, validity checking is more difficult than tautology checking in propositional logic.

2.4 High Level Symbolic Simulation

In a previous section, we identified two channels for employing high level static analysis algorithms. Both channels require reasoning with term level expressions in RTL. Symbolic simulation at the bit level is a technique that builds expressions instead of real scalar values, in a gate level circuit. Expressions at the circuit outputs are logical functions of the circuit inputs. Symbolic simulation has been applied before in the context of higher level models. We discuss symbolic simulation at the register transfer

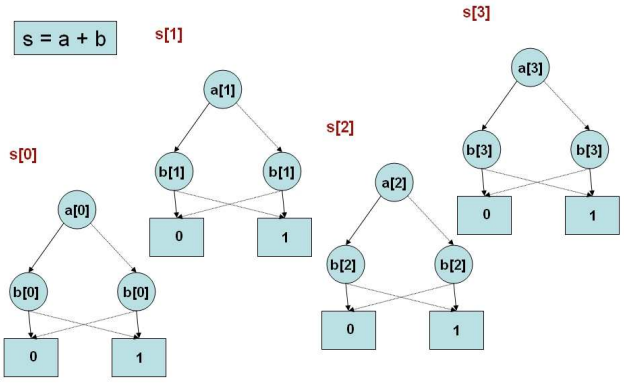


Figure 2.2. BDD representation of the state graph for the control flow graph

and behavioral levels. Since RTL has variables and operators that create different types of expressions, symbolic simulation in RTL also contains these expressions. Every variable in the RTL, other than inputs, can be symbolically simulated. A symbolic simulation for a statement in Verilog is the expression formed by transitively substituting the left hand side of the statement by the conjunction of the right hand side and the guards that can cause the statement to execute until the inputs are reached. The expressions contain all the RTL operators and variables. The symbolic simulation, then, is equivalent to stepping through, or rewriting the design algebra, one step at a time.

When applied to sequential circuits modeled by `always` blocks, the symbolic simulation needs to incorporate the looping between the end to the beginning of the block. In other words, there should be a provision for the sequentiality. As in bit level symbolic simulation, high level symbolic simulation also consists of an “unrolling” step. This basically creates multiple copies of the design, each annotated by the (relative) time in which they were computed. The circuit is unrolled with respect to the system clock, since all the circuits in synthesizable RTL are synchronous. The number of possible sequences of the unroll, therefore, are a small number. (This is in contrast to an asynchronous system, where the number

of possible unrolls could get unruly.)

Figure 2.3 gives an example Verilog code. Let us look at how the output variable `dat_o` is symbolically simulated. The first substitution for `dat_o` is where its assigned, with its right hand side variable and the guard under which the assignment statement can be executed. This guard is the false value of `tens_select`in the current time cycle, t. The value of `tens_select` in the current time cycle, $t+1$ depends on the value of `dat_i[4]` in the previous time cycle t. Transitively substituting for `tens_select` by annotating it with the time in which it was executed, gives us the expression for it. This substitution stops here, since `dat_i[4]` is an input. The resulting symbolic expression for `dat_o` is

```
dat_o(t+2) = ITE(dat_i[4](t+1), (dat_i[3:0](t) + 4'd10), dat_i[3:0](t))
```

Since this is a sequential circuit, unrolling by a single clock cycle would give us a similar symbolic expression, but one that is annotated with a different time cycle.

```
dat_o(t+1) = ITE(dat_i[4](t), (dat_i[3:0](t-1) + 4'd10), dat_i[3:0](t-1))
```

The unrolling is guaranteed to terminate for any arbitrary Verilog design. The theoretical limit for this unrolling is the synchronous nature of RTL designs, as well as the finite bit widths of the variables in the system. Since the unrolling has to stop when all the variables have reached their maximum bitwidth, this algorithm will necessarily terminate. However, practically, the algorithm is very efficient. An implementation optimization that identifies similar copies and maintains their identity with respect to time annotations, can make this algorithm very cheap.

```
module bcd_to_bin (clk_i, dat_i, dat_o);
    input  clk_i;
    input  [4:0] dat_i;
    output [3:0] dat_o;

    always @(posedge clk_i)
    begin
       tens_digit <= dat_i[3:0] + 4'd10;
       no_tens_digit <= dat_i[3:0];
       tens_select <= dat_i[4];
    end

    always @(posedge clk_i)
    begin
       if (tens_select)
           dat_bin_o <= tens_digit;
       else
           dat_bin_o <= no_tens_digit;
    end
endmodule
```

Figure 2.3. Example Verilog RTL code for BCD to binary conversion.

2.5 High level static analysis algorithms

In the remaining part of the thesis, we use the three pronged approach to solve challenging verification problems. We use high level symbolic simulation to traverse the design, domain aware analysis to identify the best verification approach for a given design, as well as efficient decision procedures to check the design.

Reasoning with the control flow graphs at the higher level, or *high level static analysis* is discussed with respect to RTL, but the discussion holds true for behavior and transaction levels. This analysis can be used in two distinct ways.

- Term level expression verification: Term level expressions can be manipulated syntactically by high level symbolic simulation. The resulting RTL terms can be checked by using a high level decision procedure like a rewriting engine. In the case of property checking, this decision procedure will be able to check the term level expressions against existing identities, and prove properties on the design. *Domain aware analysis* can be used to create the design calculus that provides identities for checking. In the case of equivalence checking, the comparison points between the two designs will be set and terms compared according to the domain aware analysis.The decision procedure in this case would be the rewriting engine.

- Reducing size of state transition graph: A slew of techniques that reason at RTL, along with the property specifications or assertions, can create smaller designs by high level symbolic simulation of variables. The design can be abstracted away with respect to the property in the case of property checking, or with respect to observable states in the case of equivalence checking. In other words, a smaller abstraction can be created, that is pertinent to the verification problem at hand. *Domain*

aware analysis can be used to create the smaller design. This smaller, tractable design can then be passed to various gate level verification algorithms.

Chapter 3

Combinational Equivalence Checking at Register Transfer Level

3.1 Introduction

Verification of integer arithmetic circuits has been a widely studied problem. Verification of relatively less complex designs like adders and shifters is an achievable target by state-of-the-art tools and techniques. However, in the case of complex arithmetic circuits, current verification methods do not scale. In particular, integer multipliers present a major challenge to verification tools and techniques. These circuits can be prohibitively expensive to analyze with Boolean-level algorithms that undertake explicit state space traversal [35, 37, 41, 60, 108, 250]. State-of-the-art combinational equivalence checkers that use state-space based techniques and Boolean reasoning cannot verify multipliers larger than 16×16 due to their large state space. Specialized techniques like Binary Moment Diagrams (BMDs) [37] can verify big word size multipliers, but are not generic in their scope. Although deductive verification techniques can handle the multiplier state space, a high level of user expertise is required to handle the engine, and considerable effort is spent in proving many ancillary lemmas before proving the main theorem. A multiplier proof, for instance, can take many person-days to accomplish.

We propose a three pronged solution using high level static analysis for formally verifying fixed-point combinational arithmetic circuits. including multipliers at the RT-level. We prove the equivalence of an implementation (or revised) Verilog RTL design against a specification (or golden) Verilog RTL design. We translate the golden and revised combinational Verilog RTL designs into Term Rewriting Systems

(TRSs). We then prove input/output equivalence between the two TRSs. This notion of equivalence is compositional and thus also affords proof decomposition through stepwise refinement. In order to decompose this proof, we compute a set of *comparison points*.

Domain aware analysis

Arithmetic circuits have sufficient structural regularity to afford analysis by functional decomposition and are, therefore, ideal candidates for our verification by stepwise refinement. The principal merit of this technique is in computing comparison points automatically. An effective heuristic is applied to compute the comparison points automatically, using the knowledge about the domain of arithmetic circuits.

Arithmetic circuits are often designed and optimized incrementally from a base design. This can afford the development of generically applicable decomposition strategies for equivalence proofs of the optimized design against the base design. We have found this to be the case for several families of integer arithmetic circuit designs and have leveraged this intuition in our tool to verify the optimized design against the base design. We have applied our technique successfully to the verification of optimized adders, comparators, shifters, and multipliers.

Since verification of multipliers provides an interesting nexus of challenge and opportunity, we make multiplier verification the main focus of this work. We split the space of arithmetic designs into standard (base) designs and the modified (optimized) designs. Standard designs are widely used designs. In multipliers, standard designs are Booth, Wallace Tree and Array multipliers. We prove the equivalence of optimizations to these designs against the standard designs and prove the equivalence of the standard designs against a simple golden Shift-and-Add multiplier. To illustrate our technique, we present the

proof of correctness of a non-trivial example, a 128×128 Booth multiplier [32] , verified against a Shift-and-Add multiplier. We also outline the steps for proving the correctness of BISMUL [249], an optimized Booth multiplier against the normal Booth multiplier. A comparison of our approach with existing Boolean equivalence checkers shows the ability to verify multipliers much larger than currently possible.

High level symbolic simulation

High level symbolic simulation is approached with a different flavor in this work. Since high level symbolic simulation is the process of stepping through the design algebra, it is similar to stepwise rewriting. In this work, we convert both RTL designs that need to be checked for equivalence into an intermediate representation, Term Rewriting Systems (TRSs) [148]. Our technique involves stepwise refinement of Term Rewriting Systems (TRSs) [148].

Decision procedure

Our tool, Verifire is a dedicated arithmetic circuit checker that provides automatic support for the proposed technique. The tool translates the golden and revised designs from Verilog source into TRSs. It automatically generates the comparison points for the incremental equivalence proof of the two TRSs. It uses an incomplete, but efficient method to generate equivalent proofs at the comparison points. Our technique retains the efficiency and the size independence of deductive verification techniques, while sacrificing automation minimally. We present a dedicated arithmetic circuit checker, as opposed to a generic rewriting engine that involves considerable user interaction. All of the analysis performed by the tool is done on *terms* composed of RTL operators (*e.g.,* bitwise-and, left-shift, etc.) as opposed to the

Boolean netlist level – the level at which equivalence checkers operate. Terms are more concise and more efficient to manipulate than netlists. The potential downside is the incompleteness of the equivalence prover, but we have found in practice, that sufficiently complete provers are reasonable to implement.

Our main contributions in this work are listed below.

- We present a dedicated arithmetic circuit checker, as opposed to a generic rewriting engine. Our tool operates on the actual RTL circuit implementation and generates proofs at that level of detail. There is minimal overhead of creating an environment, proving ancillary lemmas etc., as opposed to a generic rewriting engine, thereby yielding savings of many person-months on complex proofs.
- Our experimental results are presented on large multipliers that are very complex circuits for verification. We demonstrate that our technique can perform equivalence checking between two diverse multiplier designs, irrespective of size, thereby showing performance benefits of many orders of magnitude over widely accepted industrial methods.
- We extend our technique to verify incremental modifications or optimizations to existing designs, thereby covering a large portion of the design space.
- We present a novel decomposition strategy for RT-Level equivalence checking. In our technique, comparison points are computed automatically by the equivalence checker.
- We define and use a novel notion of decomposed TRS equivalence.
- We present a methodology to automatically determine the decomposition in the TRS equivalence proof.

Section 3.3 explains our technique in detail. We illustrate our algorithm with a non-trivial example of adders in Subsection 3.3.4. We present our tool, Verifire, in Subsection 3.3.5. We provide a theoretical definition of TRS equivalence and our proof structure in Section 3.4. In Section 3.5 we provide a detailed correctness proof for the Booth multiplier and outline the proof for BISMUL. We also provide the results of comparing our tool against commercial equivalence checkers for large multiplier designs. We discuss other multiplier verification techniques and conclude in Section 3.6.

3.2 Background

We briefly review several definitions and concepts about term rewriting. A Term Rewriting System is a set of rewrite rules where a rewrite rule is an ordered pair of terms denoted as $(t_1 \rightarrow t_2)$ with $\rightarrow$ pointing in the direction of the rewrite. A TRS is *terminating* if there are no infinite rewrite sequences $t_1 \rightarrow t_2 \rightarrow \ldots$. A TRS is *confluent* if any divergence in rewriting is eventually joined. A *normal form* is a term which cannot be rewritten any further. Termination ensures the existence of normal forms, while confluence ensures their uniqueness.

Term Rewriting Systems have been used in the past for program verification [17, 19, 99]. In the context of hardware, rewriting strategies have been used in the past to design correct circuits [26, 120, 212, 215]. Term Rewriting Systems were first proposed for hardware verification in [51]. Subsequently, they have been used for checking functional correctness of hardware [184, 251].

3.3 The Algorithm

Our goal is to prove the equivalence of an implementation and a specification design. We assume that both these designs are (or can be translated into) combinational Verilog RTL modules which define a function from their inputs to outputs. Therefore, the equivalence of these functions is the target of the analysis. While traditional combinational equivalence checking tools also analyze the same problem at the gate level, our analysis is at the RT-level.

The monolithic verification problem is intractable in general and we use signal[1] names in the two modules as guidance in decomposing this equivalence proof.

[1]We use the word *signal* to refer to Verilog variables in RTL modules, and reserve *variable* for variables in TRSs

```
main (vG, vR) {
    trsG := translate (vG)
    trsR := translate (vR)
    proof_outcome := prove (trsG, trsR)
}

prove (trsG, trsR) {
    CP := computeComparePoints (trsG, trsR)
    for (every comparison point (cG, cR) ∈ CP)
        if (reduce (cG) is not equal to reduce (cR))
            return failure
    return success
}

reduce (t) {
    while (some rule can be applied)
        rewrite (t)
}
```

Figure 3.1. The Algorithm.

The algorithm for our technique is presented in Figure 3.3. The golden and revised Verilog designs are represented by *vG* and *vR* respectively. A mapping of input and output signal names between the two designs is provided.

The *translate()* function translates the Verilog into a TRS – the resulting TRS will be termed a *structural* TRS. The details of the translation of Verilog designs to structural TRSs can be found in Subsection 3.3.1. *trsG* and *trsR*, therefore, represent the structural TRSs of the corresponding Verilog design. The *prove* function is invoked for checking equivalence of the two TRSs. We decompose the equivalence proof using *comparison points* for matching the TRSs. The function that computes these comparison points is *computeComparePoints()*(explained later in Subsection 3.3.2). For each comparison point *cG*

in the golden TRS, the corresponding point in the revised TRS is *cR*. The *reduce()* function simplifies the terms at the comparison point by applying a set of rewrite rules. The simplification is done until no more rules can be applied. If the simplified terms *reduce(cG)* and *reduce(cR)* are found to be equal, the process is repeated at the next comparison point until all comparison points have been proven equal.

3.3.1 Verilog to TRS translation (translate())

Our approach begins with the translation of the source Verilog modules into a *structural* TRS which "simulates" the Verilog evaluation semantics *i.e* it exactly follows the behavior and evaluation semantics of the Verilog design. As such, the structural TRS is a syntactically translated entity that is at the same level of abstraction as the Verilog design. The result of this translation is a rewrite system which can be used to compute the symbolic term for any signal in terms of other signals and/or primary inputs. Each hierarchical signal is represented by a new constant function symbol (*signal function*). Therefore the hierarchy is "flattened" in this structural TRS. Rewrite rules rewrite each signal function into an expression consisting of RTL operators and other signal functions. Consider the example translation of the following Verilog into a structural TRS:

```
module top(inA, inB, opt, sel, out);
  input inA, inB, opt, sel;
  output out;
  reg out;
  wire fout;
  foo f1 (fout, inA, inB);
  always@* begin
    out = sel ? fout : opt;
```

```
    out = inA | out;
  end
endmodule

module foo (S, A, B);
  input A, B;
  output S;
  wire S;
  assign S = A ∧ B;
endmodule
```

The resulting structural TRS for the module `top` is the following:

```
f1.A() → inA()
f1.B() → inB()
f1.S() → (f1.A() ∧ f1.B())
fout() → f1.S()
out1() → if(sel(),fout(),opt())
out2() → (inA() | out1())
```

Since the signal `out` is assigned more than once, the different assignments are split into two signal functions `out1` and `out2`. We assume for simplicity in the Verilog modules, that primary inputs are never assigned, and primary outputs are never referenced.

We assume that the input Verilog is race-free (*i.e.* no multiple parallel assignments for the same signal), and loop-free (*i.e.* no cyclic dependencies between combinational `always` blocks). The resulting structural TRS will then be convergent, *i.e.* confluent (due to race-free assumption) and terminating (due to loop-free assumption). The loop-free and race-free assumptions can be checked by standard Verilog

linting tools. Note that for this structural TRS, the Verilog RTL operators are uninterpreted; the structural TRS is only used to construct terms defining the values of signals in terms of other signals.

3.3.2 Computing comparison points (computeComparePoints())

The *computeComparePoints()* function finds the intermediate comparison points using a heuristic. We have mentioned that the structural TRS can be used to find the symbolic term for any signal in the design. If all the bits of the signal are assigned together (in the same rewrite step), there will be a single corresponding symbolic term for that signal. However, if the bits of the signal are assigned separately (different rewrite steps), there will be more than one symbolic term for the signal. We clarify this with an example.

Consider a 32-bit multiplier that we would like to verify, which has `mul_result[31:0]` as an output signal. If the multiplier RTL has only one assignment statement assigning the entire value `mul_result[31:0]`, then there is only one rewrite step which results in generating the symbolic term for `mul_result`. Therefore, there will be a single symbolic term for `mul_result`.

Assume the multiplier's RTL description is written such that 8 bits of the output are assigned a value together, *i.e* at the same time. All the 32 bits of the output are, therefore, assigned values after 4 such assignments. In the TRS for this multiplier, there will be 4 rewrite steps corresponding to these 4 assignments. Each rewrite step generates a symbolic term for `mul_result`. Hence, there will be 4 symbolic terms that correspond to `mul_result[7:0]`, `mul_result[15:8]`, `mul_result[23:16]`, and `mul_result[31:24]`.

Every subset of bits assigned, will thus have its symbolic term. We will call such assignments (to different subsets of bits of the same signal), *reassignments* in the rest of the paper. Thus, a set of

reassignments for a signal define a partition of the bits for the signal. In our example, the 4 *reassignments* define the partition {`[31:24], [23:16], [15:8], [7:0]`} on the 32 bits of the output signal `mul_result`.

Let the golden multiplier design be a shift-and-add design which has `golden_mul_result` as an output signal . The RTL description for this design assigns a value to the output 1 bit at a time. Therefore, in the TRS of the golden design, there will be 32 *reassignments* defining the partition {`31,30, ...2,1,0`} on the bits of the signal `golden_mul_result`.

Comparison points are computed for every (declared) output signal in the two designs in the following way.

1. The reassignment bit partitions in the golden and revised design are computed. In our example, the golden partition is {`31,30, ...2,1,0`} and the revised partition is {`[31:24], [23:16], [15:8], [7:0]`}.

2. A new variable is defined for every set of bits in the pairwise intersection of these two partitions. In our example, the pairwise intersection groups entries of the golden partition together. The new variables in the golden design will be `mG1, mG2, mG3,` and `mG4` corresponding to the bit sets {`7,6,...1,0`}, {`15,14,...9,8`}, {`23,22,...17,16`}, and {`31,30,...25,24`} respectively. The new variables in the revised design will be `mR1, mR2, mR3,` and `mR4` corresponding to the bit sets {`[31:24], [23:16], [15:8], [7:0]`} respectively.

3. The new variables obtained are paired to define the set of comparison points. In our example, the set of comparison points is {`(mG1, mR1), (mG2, mR2), (mG3, mR3), (mG4, mR4)`}.

We thus compute a partition of the bits for a particular output defined by the reassignments in both the

golden and revised designs. This is a simple heuristic that appears to work well for arithmetic circuits with common outputs and possibly some common internal points.

It is important to note that these comparison points are computed before the equivalence checking process by statically analyzing the two RTL descriptions. They do not form a part of the equivalence checking algorithm itself. These comparison points are used for decomposing the equivalence checking space, to create smaller, more tractable problems to be proven by the equivalence checking technique, in our case, term rewriting. The scalability of the algorithm, therefore, does not depend on the computation of comparison points.

Traditional combinational equivalence checkers routinely face the issue of not being able to reliably conclude that two designs are not equivalent. This is the problem of *false negatives*. In order to mitigate the verification complexity, equivalence checkers perform *hierarchical verification* that comprises isolated verification of each hierarchical block, under the assumption that exact functional equivalence at hierarchical boundaries is preserved. False negatives occur in such hierarchical verification when either (a) functional equivalence at hierarchical boundaries is not preserved, or (b) when the block of design is functionally equivalent only when is constrained by the environment, not with unconstrained variables as viewed by the hierarchical verification process [16]. These issues are circumvented by our technique, since we "flatten" the hierarchy during translation of the design from Verilog to TRS, as shown in the example is Subsection 3.3.1 as well as the RCA and CLA examples in Subsection 3.3.4. We verify the designs without maintaining their hierarchical boundaries, and therefore do not encounter the false negative problem in our technique.

Another noteworthy difference between our technique and traditional gate level equivalence checkers is that we do not view each internal register as a comparison point. Our comparison points are the assignments or reassignments to the output signals of the design. Consequently, between arithmetic designs

whose (output) size and number of outputs are equivalent, a correspondence between the comparison points in the two designs can always be expected.

3.3.3 Checking equivalence of terms (reduce())

The *reduce()* function checks for equivalence between two symbolic terms by rewriting based simplification. This is achieved by using a separate database of rewrite rules that codify various identities about the RTL operators. For example, one may introduce an absorption and association rule for &, as well as rules for reducing arithmetic and left-shifts:

$$(x \,\&\, x) \;\rightarrow\; x \tag{3.1}$$
$$((x \,\&\, y)\&z) \;\rightarrow\; (x \,\&\, (y \,\&\, z)) \tag{3.2}$$
$$(x << 3) \;\rightarrow\; (x << 2) + (x << 1) + (x << 1) \tag{3.3}$$
$$((x << 1) - x) \;\rightarrow\; x \tag{3.4}$$
$$((x << 1) << 1) \;\rightarrow\; (x << 2) \tag{3.5}$$

These rules can be generic or design specific, as demonstrated in Section 3.5. At any compare point, while trying to prove equivalence, the *reduce()* function selects a set of rewrite rules from the database and applies them in some order. If the resulting simplifications fail to prove term equivalence, a different set (and/or ordering) of rewrite rules is chosen and applied. These *proof iterations* terminate when term equivalence is established, or when no more rules can be applied.

When two designs are declared unequal by our technique, the result is reliable, unless the rules in the rule base are not sufficient to simplify the terms under comparison. In that case, the prover part of our

tool gives the proof trace at the last comparison point. User intervention is required to check the proof for more rules, or accept the negative result.

However, it can be inferred that two terms are equivalent if their simplified values are equal. We will discuss this term reduction function in a later section, but we note that the procedure is not complete in determining term equivalence. Instead *reduce()* is designed to be efficient and sufficient for the domain of circuits to be analyzed. In cases where the set of rules used by *reduce()* is insufficient, we allow the user to augment this set. The decomposition of the equivalence check using comparison points and incremental refinement lessens the requirements on efficiency as well as sufficiency for the function *reduce()*.

3.3.4 Example: Adder verification

We present an illustrative example of how our technique works on adder circuits. We verify a 16-bit Carry Lookahead Adder (CLA) design. We use a simple Ripple Carry Adder (RCA) as the golden design for adders. It adds two vectors by doing a bitwise xor and generates a corresponding carry bit. The Verilog code for a 16-bit RCA design is shown in Figure 3.3.4.

The structural TRS is obtained by applying the *translate()* function to the RCA. The structural TRS for the module `rca16bit` is the following:

```
R1: rca1bit0.A()     → A[0]()
    rca1bit0.B()     → B[0]()
    rca1bit0.C()     → Cin()
    rca1bit0.S()     → (rca1bit0.A() ∧ rca1bit0.B() ∧ rca1bit0.C())
    rca1bit0.Cout()  → ((rca1bit0.A() & rca1bit0.B()) |
```

```
module rca16bit(A, B, Cin, S, Cout);
input [15:0] A, B;
input Cin;
output [15:0] S;
output Cout;
reg S, Cout;
wire [14:0] Carry;

rca1bit rca1bit0(A[0], B[0], Cin, S[0], Carry[0]);       R1
rca1bit rca1bit1(A[1], B[1], Carry[0], S[1], Carry[1]);  R2
                ⋮
rca1bit rca1bit15(A[15], B[15], Carry[14], S[15], Cout); R16
endmodule

module rca1bit(A, B, C, S, Cout);
input A, B, C;
output S, Cout;
assign S = A ∧ B ∧ C;
assign Cout = A&B | B&C | C&A;
endmodule
```

Figure 3.2. RCA Verilog.

```
                        (rca1bit0.B() & rca1bit0.C()) |
                        (rca1bit0.C() & rca1bit0.A()))
    S[0]()          → rca1bit0.S()
    Carry[0]()      → rca1bit0.Cout()

R2: rca1bit1.A()    → A[1]()
    rca1bit1.B()    → B[1]()
    rca1bit1.C()    → Carry[0]()
    rca1bit1.S()    → (rca1bit1.A() ∧ rca1bit1.B() ∧ rca1bit1.C())
    rca1bit1.Cout() → ((rca1bit1.A() & rca1bit1.B()) |
                        (rca1bit1.B() & rca1bit1.C()) |
```

```
                          (rca1bit1.C() & rca1bit1.A()))
    S[1]()           → rca1bit1.S()
    Carry[1]()       → rca1bit1.Cout()
       .
       .
       .
R16:rca1bit15.A()    → A[15]()
    rca1bit15.B()    → B[15]()
    rca1bit15.C()    → Carry[14]()
    rca1bit15.S()    → (rca1bit15.A() ∧ rca1bit15.B() ∧ rca1bit15.C())
    rca1bit15.Cout() → ((rca1bit15.A() & rca1bit15.B()) |
                          (rca1bit15.B() & rca1bit15.C()) |
                          (rca1bit15.C() & rca1bit15.A()))
    S[15]()          → rca1bit15.S()
    Cout()           → rca1bit15.Cout()
```

We observe that labels **R1**...**R16** in Figure 3.3.4 corresponds to the set of rules `R1`...`R16` in the structural TRS.

The target design, a Carry Lookahead Adder (CLA) is similarly translated from its Verilog implementation to a structural TRS. The Verilog code for the CLA is given in Figure 3.3.4.

There are four module calls to the `cla4bit` module by the main module. There are four `cla4bit` blocks in the design. The `cla4bit` module computes four successive carries at a time. The sum for the corresponding four bits is calculated in this module. The `cla4bit` module also calls the `PGgen` module, that generates the `P`s(propagated carries) and the `G`s (generated carries) for the block. A module called `fastcarry` is called to calculate the input carry values `(Cin, C[3], C[7], C[11])` for

```
module cla16bit (A, B, Cin, S, Cout);
 input [15:0] A, B;
 input Cin;
 output [15:0] S;
 output Cout;
 reg S, Cout;
 wire C3, C7, C11;
 fastcarry fc (A, B, Cin, C3, C7, C11);     R1
 cla4bit cla0 (A[3:0], B[3:0],Cin, S[3:0]); R2
 cla4bit cla1 (A[7:4], B[7:4], C3, S[7:4]); R3
 cla4bit cla2 (A[11:8],B[11:8],C7,S[11:8]); R4
 cla4bit cla3 (A[15:12], B[15:12], C11, S[15:12]); R5
endmodule

module cla4bit (a, b, cin, s);
 input [3:0] a, b;
 input cin;
 output [3:0] s;
 wire [3:0] c;
 assign c[0] = g[0] | p[0]&cin;
 assign c[1] = g[1] | g[0]&p[1] | p[1]&p[0]&cin;
 assign c[2] = g[2] | g[1]&p[2] | g[0]&p[2]&p[1] | p[2]&p[1]&p[0]&cin;
 assign c[3] = g[3] | g[2]&p[3] | g[1]&p[3]&p[2]
           | g[2]&p[3]&p[2]&p[1] | p[3]&p[2]&p[1]&p[0]&cin;
 assign s[0] = a[0] ^ b[0] ^ c[0];
 assign s[1] = a[1] ^ b[1] ^ c[1];
 assign s[2] = a[2] ^ b[2] ^ c[2];
 assign s[3] = a[3] ^ b[3] ^ c[3];
 PGgen pg0 (a[0], b[0], p[0], g[0]);
 PGgen pg1 (a[1], b[1], p[1], g[1]);
 PGgen pg2 (a[2], b[2], p[2], g[2]);
 PGgen pg3 (a[3], b[3], p[3], g[3]);
endmodule

module PGgen (a, b, p, g);
 input a, b;
 output p, g;
 assign p = a ^ b;
 assign g = a & b;
endmodule

module fastcarry (a, b, cin, c3, c7, c11);
/*Accelerated Carry Computation -- not detailed.*/
endmodule
```


Figure 3.3. CLA Verilog.

each of the four `cla4bit` blocks.

The structural TRS for the module `cla16bit` is the following (not complete):

```
R1:  C3()          → fc.c3()
     C7()          → fc.c7()
     C11()         → fc.c11()
        .
        .
        .
R2:  cla0.a[0]() → A[0]()
     cla0.b[0]() → B[0]()
     cla0.s[0]() → (cla0.a[0] ∧ cla0.b[0] ∧ cla0.c[0])
     cla0.a[1]() → A[1]()
     cla0.b[1]() → B[1]()
     cla0.s[1]() → (cla0.a[1] ∧ cla0.b[1] ∧ cla0.c[1])
     cla0.a[2]() → A[2]()
     cla0.b[2]() → B[2]()
     cla0.s[2]() → (cla0.a[2] ∧ cla0.b[2] ∧ cla0.c[2])
     cla0.a[3]() → A[3]()
     cla0.b[3]() → B[3]()
     cla0.s[3]() → (cla0.a[3] ∧ cla0.b[3] ∧ cla0.c[3])
     cla0.cin()  → Cin()
     S[3:0]()    → cla0.s[3](), cla0.s[2], cla0.s[1](), cla0.s[0]
        .
        .
        .
R5:  cla3.a[0]() → A[12]()
     cla3.b[0]() → B[12]()
     cla3.s[0]() → (cla3.a[0] ∧ cla3.b[0] ∧ cla3.c[0])
```

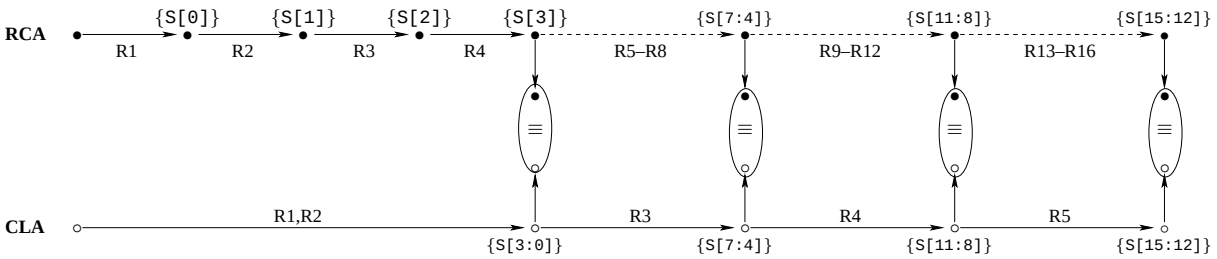


Figure 3.4. Proof of correctness of the Carry Lookahead adder compared against the Ripple Carry Adder. • represents terms of the RCA. ∘ represents terms of the CLA. The variable within {} is the reassigned output. ≡ represents the equivalence between the outputs at each comparison point.

```
cla3.a[1]() → A[13]()
cla3.b[1]() → B[13]()
cla3.s[1]() → (cla3.a[1] ∧ cla3.b[1] ∧ cla3.c[1])
cla3.a[2]() → A[14]()
cla3.b[2]() → B[14]()
cla3.s[2]() → (cla3.a[2] ∧ cla3.b[2] ∧ cla3.c[2])
cla3.a[3]() → A[15]()
cla3.b[3]() → B[15]()
cla3.s[3]() → (cla3.a[3] ∧ cla3.b[3] ∧ cla3.c[3])
cla3.cin()  → Cin()
S[15:12]()  → cla0.s[3](), cla0.s[2], cla0.s[1](), cla0.s[0]
```

Labels **R1**...**R5** in Figure 3.3.4 correspond to the sets of rules `R1`...`R5` in the structural TRS.

We are interested in showing equivalence with respect to the primary output signals `S` and `Cout`. The *computeComparePoints()* function now calculates the comparison points. The signals `S` and `Cout` are reassigned in the design. If the same subset of bits are reassigned in both designs, it is recognized as a

comparison point. In the RCA, every rewriting step assigns a value to a single bit of the output signals. For instance, `R1` in the RCA represents a single rewrite step that assigns a value to `S[0]`. However, in the CLA, every rewriting step, (*i.e.*, `R2...R5` in the TRS for CLA) assigns a value to four bits of the sum (*i.e* `S[3:0]...S[15:12]`). Therefore, a single step in CLA corresponds to four steps in the RCA. A comparison point is recognized at `S[3]`. The symbolic terms obtained in both designs for `S[3]` are compared.

The *reduce()* function proves equivalence at the first comparison point. It is simple to understand how the symbolic terms generated are equivalent, by tracing the rewrite steps in both the TRSs that lead to the (reassigned) outputs. A correspondence between the rules for the two TRSs is given below. Figure 3.4 explains this in an intuitive manner.

Applying rules `R1...R4` in the RCA TRS corresponds to rule `R2` in the CLA TRS, since the four bits of the sum are computed as an xor of the corresponding input operand and carry bits. However, the input carry terms in the two TRSs are different. The symbolic term for `Carry[3]` in the RCA is obtained by applying rules `R1...R4`. The corresponding term in the CLA TRS, `c[3]`, is obtained from rule `R2`. Applying rule `R2` of the CLA TRS initially gives this fourth stage carry in terms of the `P`s and `G`s of previous stages, and subsequently gives the same expression in terms of `A` and `B`, instead of `P`s and `G`s. Therefore, the symbolic values of the carry terms in both the TRSs turn out to be exactly equal.

Once `S[3]` is verified, the same procedure is repeated at the remaining comparison points, namely `S[7], S[11]` and `S[15]`. The normal form of both the TRSs is reached when `S[15]` is computed. The two TRSs are thereby proved equivalent.

3.3.5 Verifire : **An automated proof generator**

Our tool, Verifire automates the algorithm introduced in Section 3.3. Vtranslate automates the *translate()* function and Vprove automates the *prove()* function of the algorithm.

Vtranslate is a compiler which accepts Verilog (synthesizable by commercial tools) as input. It automatically identifies and flattens the module hierarchy and constructs the structural TRS for the entire circuit. Vprove automatically generates equivalence proofs between two TRSs using the technique of stepwise refinement, outlined in Section 3.3. The golden TRS and the revised TRS are inputs to the proof engine. The tool automatically generates a proof, or returns an error trace if it cannot establish the proof.

Vprove implements the *computeComparePoints()* and *reduce()* functions. The *reduce()* function is called as and when each comparison point is computed. The tool keeps track of potential multiple reduction rules and implements a backtracking algorithm in order to establish the equivalence at each comparison point.

Verifire was implemented in C++ and was used to prove many multiplier circuits. The tool can automatically generate proofs for standard multiplier designs like the *Booth* multiplier, *Array* multipliers and *Tree* multipliers. It can also automatically generate proofs for multiplier designs that are modifications of these standard designs.

3.4 TRS equivalence

This section provides a technical definition of TRS equivalence and proof that the decomposition of TRS equivalence we use (via comparison points) is sound. This section is not requisite to the main points

of this paper and could be skipped by the reader if desired.

Given a specified top module M of a Verilog design, the *primary inputs* $PI(M)$ are the signal functions for the primary input signals of M, and the primary outputs $PO(M)$ are the corresponding signal functions for the primary output signals.

Using the structural TRS for top module M, we can define the function $\Phi_M(s, X)$ which takes a signal function s and a set of signal functions X and returns the normal form of s in the rewrite system R minus the rules for rewriting functions in $X \setminus \{t\}$.

We need to define the notion of equivalence we wish to check. For a term t we define the $support(t)$ to be the set of signal functions and variables in t. For two terms s and t we define $s \equiv t$ as $support(s) = support(t)$ and for all ground substitutions σ of $X = support(s)$, we have $s[X/\sigma] = t[X/\sigma]$[2] Given modules G and R (the golden and revised modules where $PI = PI(G) = PI(R)$ and $PO = PO(G) = PO(R)$) and a set of signal functions X, define $R \sim_X G$ as $(\forall t \in X : \Phi_G(t, X) \equiv \Phi_R(t, X))$. Note that in order for this definition to be useful we assume that *equivalent* signals in the two modules G and R have the same signal name. In cases of reassignments to the same signal in either module, we may need to adjust the names assigned to the reassigned signals in order to ensure correspondence.

We wish to prove $R \sim_{PO} G$. As explained earlier, we decompose the proof by computing a set of *comparison point* signal functions C. We make use of the following property to transfer the result to the proof of equivalence for PO:

Theorem 3.4.1. $\forall R, G, C, O : (((R \sim_C G) \wedge (O \subseteq C)) \Rightarrow (R \sim_O G))$

[2]We use the notation $t[X/\sigma]$ to denote the term t where variables in X have been replaced according to the assignment σ. If σ does not assign to any variables in set X then the term t is unchanged.

Proof Outline. Take an arbitrary signal function $s \in O$. We first observe that the term $\Phi_R(s, O)$ is equal to the iterative expansion beginning with the term $\Phi_R(s, C)$ where in each step, the signals $x \in C \setminus O$ are substituted by the corresponding terms $\Phi_R(x, C)$. A similar observation holds for $\Phi_G(s, O)$. Then for any ground substitution for O, one can prove by induction following the iterative expansions we observed, that the desired equality holds between $\Phi_R(s, O)$ and $\Phi_G(s, O)$ using the assumption $R \sim_C G$ to relieve the induction hypothesis and substituting equals for equals along the way.

Thus, we prove $R \sim_{PO} G$ by proving $R \sim_C G$ instead where $PO \subseteq C$. The following additional (trivial) property is useful in chaining $\sim$ proofs together:

Theorem 3.4.2. $\forall R, I, G : (((R \sim_{PO} I) \wedge (I \sim_{PO} G)) \Rightarrow (R \sim_{PO} G))$

Our procedure for proving $R \sim_{PO} G$ consists of the following two steps, compute a set of *comparison points* C and then prove $R \sim_C G$.

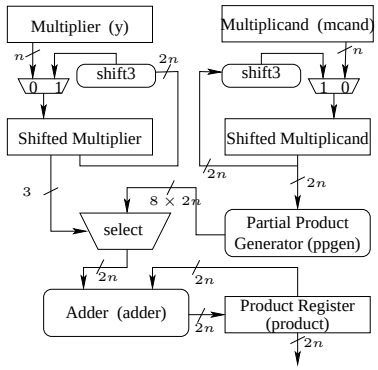
In order to check $R \sim_C G$, we iterate through each signal $x \in C$, and compute $s = \Phi_R(x, C)$, compute $t = \Phi_G(x, C)$, and check if $s \equiv t$. The mechanism for checking $s \equiv t$ for two given terms is the simplification function $reduce(t)$, which maps a term t to a reduced term which is equal under all substitutions to t. If $reduce(s) = reduce(t)$, we can deduce that $s \equiv t$.

3.5 Multiplier Verification

We consider the space of multipliers divided into *standard* and *non-standard* multipliers. The standard multipliers are the widely used, common multiplier designs like Booth, Wallace tree, Dadda Tree and Array multipliers. The non-standard multipliers have incremental optimizations made to these standard multiplier designs. We have extended our technique to cover the space of these two categories of

multipliers. We illustrate our technique on the Booth multiplier and BISMUL, an optimization of the Booth multiplier.

3.5.1 Booth Multiplier



(a)

multiplier bits	partial product generated x : multiplicand
0 0 0	pp0: 0
0 0 1	pp1: x
0 1 0	pp2: $x << 1$
0 1 1	pp3: $x << 1 + x$
1 0 0	pp4: $x << 2$
1 0 1	pp5: $x << 2 + x$
1 1 0	pp6: $x << 2 + x << 1$
1 1 1	pp7: $x << 3 - x$

(b)

Figure 3.5. (a) Architecture of a Booth multiplier. (b) Partial product terms of the Booth multiplier.

The multiplier in Figure 3.5.1(a) is a 64-bit, radix-3, non-overlapping Booth multiplier. The `ppgen` block generates the eight partial products that form the Booth encoding. The `ppsel` block selects the relevant partial product depending on the incoming bits from the multiplier. The partial products are added in the adder block. The `ppgen` is given the shifted multiplicand as input (`shift3`) to generate the current partial product. This method is repeated for all the bits of the multiplier. The result appears in the `product` register.

To prove the functional correctness of the above design, we follow the technique explained in Section 3.3. We illustrate the proof using the outline of the proof provided in that section.

We use a simple Shift-and-Add multiplier as the reference TRS for multipliers. It performs multiplication by generating partial products. It shifts the multiplicand left by one bit after every partial product

calculation. The partial product of the current stage is set to the sum of the previous partial product and the shifted multiplicand of the current stage or 0, depending on whether the multiplier bit corresponding to the current stage is 1 or 0. The Verilog code of the Shift-and-Add calls a `shift` and an `add` module iteratively.

The target design here is the Booth multiplier discussed above. Vtranslate extracts its TRS from the Verilog code.

In the case of the Booth multiplier, the *PO* needed to prove $(R \sim_{PO} G)$ is `product` as explained in Section 3.3. A sketch of this proof (as output by the tool) follows.

The first comparison point in the proof is after 3 bits of output (`product`) are updated in both TRSs. This is because the Booth updates 3 bits of its `product` simultaneously as opposed to Shift-and-Add that updates its `product` sequentially. The output of the tool after the first comparison point is as follows. `Stage` i in the tool output represents the i-th update of `product`.[3]

```
Comparison Point 1:
Reference Model: Shift-and-Add
Stage 1.
Rule x: product = product + mcand                if( y[0])
Rule y: product = product + 0                    if(  y[0])
Stage 2.
Rule x: product = product + mcand<<1              if( y[1])
```

[3]The rules in the output are reproduced in pseudo-Verilog syntax and they correspond to the rewrite rules of the TRS as described in Section 3.3. For example, rules `Reference.Stage 1.Rule x` and `Reference.Stage 1.Rule y` in the TRS would be the rewrite rule

`product()` $\rightarrow$ `product() + if (y[0](), mcand(), 0)`.

```
Rule y: product = product + 0                    if(  y[1])
Stage 3.
Rule x: product = product + mcand<<2             if( y[2])
Rule y: product = product + 0                    if(  y[2])
Revised Model: Booth
Stage 1.
Rule a: product = product + 0          if (  y[0] &   y[1] &   y[2])
Rule b: product = product + mcand      if ( y[0] &   y[1] &   y[2])
Rule c: product = product + mcand<<1   if (  y[0] &  y[1] &   y[2])
Rule d: product = product + mcand<<1 + mcand
                                       if ( y[0] &  y[1] &   y[2])
Rule e: product = product + mcand<<2   if (  y[0] &   y[1] &  y[2])
Rule f: product = product + mcand<<2 + mcand
                                       if ( y[0] &   y[1] &  y[2])
Rule g: product = product + mcand<<2 + mcand<<1
                                       if (  y[0] &  y[1] &  y[2])
Rule h: product = product + mcand<<3 - mcand
                                       if ( y[0] &  y[1] &  y[2])
```

The expressions generated from both the TRSs from the first comparison point are displayed with their corresponding rules. For instance, `Reference.Stage 1.Rule x` is `product = product + mcand if (y[0])`.

Correspondence at the comparison point is established by a case-by-case analysis of the rules. Every encoding of the Booth multiplier is compared to the Shift-and-Add, with the same conditions. For instance, `Revised.Stage 1.Rule a` gives the expression for the partial product generated for the

Booth encoding 000. Applying the condition `y[0]y[1]y[2] = 000` on the Shift-and-Add, corresponds to rules `Reference.Rule 1y`, `Reference.Rule 2y`, and `Reference.Rule 3y`. Such an analysis is performed for all cases.

The *reduce()* function is applied by Vprove to simplify corresponding terms at `Comparison Point 1`. An outline of this simplification as output by the tool is given below.

```
Correspondence (after case analysis):
  Revised             Reference
  Rule 1a  ==  Rule 1y, Rule 2y, Rule 3y
  Rule 1b  ==  Rule 1x, Rule 2y, Rule 3y
  Rule 1c  ==  Rule 1y, Rule 2x, Rule 3y
  Rule 1d  ==  Rule 1x, Rule 2x, Rule 3y
  Rule 1e  ==  Rule 1y, Rule 2y, Rule 3x
  Rule 1f  ==  Rule 1x, Rule 2y, Rule 3x
  Rule 1g  ==  Rule 1y, Rule 2x, Rule 3x
  Rule 1h  ==  Rule 1x, Rule 2x, Rule 3x
```

The set of rules used by *reduce()* is not complete, which renders establishing this correspondence non-trivial. In many situations the user might need to add to this set of rules to assist Vprove in establishing the correspondence. We illustrate this with the example of `Rule 1h == Rule 1x, Rule 2x, Rule 3x`.

`Revised.Rule 1h` under the condition `y[0]y[1]y[2] = 111` states that:

$$\texttt{product} \rightarrow \texttt{product + mcand<<3 - mcand}$$

The corresponding rules in the reference model induced by the condition are `Reference.Rule 1x`, `Reference.Rule 2x`, and `Reference.Rule 3x`. Combining the three reference rules we obtain the term:

```
product + mcand + mcand<<1 + mcand<<2
```

Applying *reduce()* on the revised term we obtain the term:

```
product + (mcand<<2 + mcand<<1 + mcand<<1) - m
```

However if the database has no further rules to invoke *reduce()*, then we cannot establish the correspondence. Hence the user needs to add a rule to the database to help simplify the term further. In this case, we add the rule

$$\texttt{((x + y) - z)} \rightarrow \texttt{(x + (y -z))}$$

With the addition of this rule, the revised term now is reduced to

```
product + (mcand<<2 + mcand<<1) + (mcand<<1 - m)
```

which is further reduced to

```
product + mcand<<2 + mcand<<1 + mcand
```

Thus we establish the correspondence.

The proof now proceeds to subsequent comparison points iteratively till the output is obtained in its normal form. Therefore $R \sim_C G$ is proved. Figure 3.6 explains the proof in an intuitive manner.

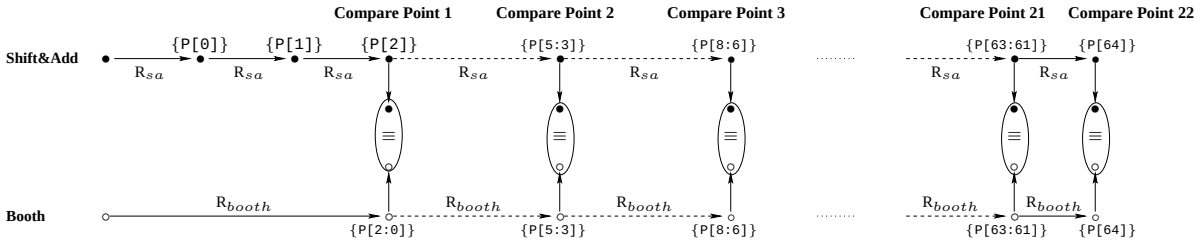


Figure 3.6. Proof of correctness of the Booth multiplier compared to the Shift&Add multiplier. • represents terms of the Shift&Add multiplier. ○ represents terms of the Booth multiplier. R_{sa} represents the rules of the Shift&Add multiplier (`Rule x` and `Rule y`). R_{booth} represents the corresponding Booth multiplier rules (`Rule a ...Rule h`). The variable `product` is the reassigned output. $\equiv$ represents the equivalence between the outputs at each comparison point.

3.5.2 BISMUL

In order to improve the performance of multipliers, more complicated algorithms and designs are used. We consider a high-performance multiplier, BISMUL [249], that is a modification of the Booth multiplier. A radix-3 Booth multiplier architecture using 3-bit scan with no overlap invariably generates dummy bits in the last 3-bit-scan. In BISMUL, this last 3-bit-scan is moved to the first 3-bit scan, so that the dummy bits can be used for odd multiple generation. The sequence for bit scanning is shown in Figure 3.8. The improvement in the multiplication speed in BISMUL is obtained by executing several Partial Product Selectors (PPSELs) in parallel. The shifting sequence of the multiplier decides the inputs to the PPSELs. These selected partial products are summed through carry-save additions.

The architecture of a 64×64 bit multiplier is shown in Figure 3.7. This architecture comprises Product Registers (PR), Partial Product Generators (PPG), Partial Product Selectors (PPS), Multiplexers and a Carry Save Adder (CSA). PPG is implemented according to Figure 3.8. PPS consists of four PPSELs. Each PPSEL has 16-bit inputs, whose sequence is shown in Figure 3.8. The operation of the BISMUL

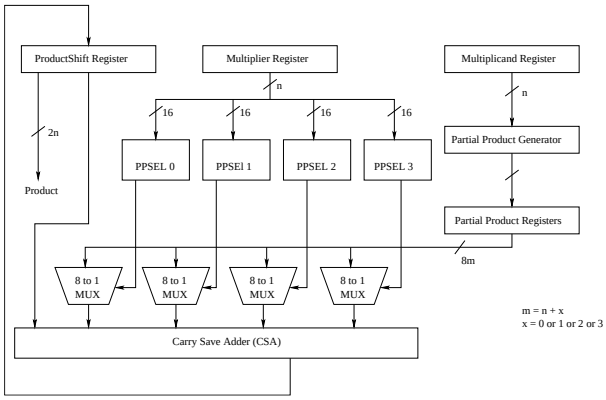


Figure 3.7. Architecture of a BISMUL.

is as follows. In the first cycle, PPG generates eight partial products and each PPSEL selects one partial product. The two dummy bits in the lower bit position in the first three bits of PPSEL cause the selection of either 0, or four times the multiplicand (000 or 100). The partial products are added and stored in the PR. The partial products get generated in the first cycle. Subsequent cycles perform the same operation as described.

We prove the BISMUL correct by using the following technique. We perform a series of reductions on the BISMUL to reduce it to its standard design, the Booth multiplier. The standard Booth multiplier is already verified using the above technique. Hence, the given non-standard design can be proven correct.

Verifire extracts the corresponding TRSs from the BISMUL and Booth Verilog code. The tool compares the modules in the non-standard design (that derive from the modules in the standard design) to the corresponding modules in the standard design. The correspondence (and equivalence) between these "derived" modules of the non-standard design, and the modules in the standard design is established by the same method as described in Subsection 3.5.1 between the Booth and the Shift-and-Add.

bit	Generation of Partial Product Terms
0 0 0	P0: 0
0 0 1	P1: multiplicand
0 1 0	P2: shift multiplicand left by 1
0 1 1	P3: add P1 and P2
1 0 0	P4: shift multiplicand left by 2
1 0 1	P5: add P1 and P4
1 1 0	P6: shift P3 to the left by one
1 1 1	P7: subtract P1 from 8

PPSEL	Inputs
PPSEL0	Multiplier[54:52],[42:40],[30:28],[18:16],[6:4][0]
PPSEL1	Multiplier[57:55],[45:43],[33:31],[21:19],[9:7][1]
PPSEL2	Multiplier[60:58],[48:46],[36:34],[24:22],[12:10][2]
PPSEL3	Multiplier[63:61],[51:49],[39:37],[27:25],[15:13][3]

Figure 3.8. The partial product terms in a BISMUL and the inputs of each PPSEL

In order to prove the reduction of BISMUL to Booth, it is enough to prove that the changed (terms) modules in BISMUL are equivalent to the original Booth terms. In this case, the terms {`ppsel0, ppsel1, ppsel2, ppsel3, mux8to1`} of the BISMUL form the revised design. The terms {`ppsel, shift3`} of the Booth act as the corresponding reference design. Similarly, the terms {`productshift, carrysaveadder`} of BISMUL correspond to the {`ppsel,adder`} terms of Booth. Therefore, it is sufficient to prove the validity of each correspondence.

We have verified the BISMUL using our technique. We have also verified the Wallace Tree multiplier. The Booth multiplier we used was designed a Booth-recoded Array multiplier. We have also verified a Dadda Tree multiplier using our technique. For each of these, we can also verify some modifications to the standard designs.

The terms in the TRS for the modified design are simplified to terms in the TRS for the standard

design. The simplification is performed using the database of rules in Vprove. This set of rules is not exhaustive and may require manual intervention when presented with an entirely new design that does not build on the standard ones. However, for a large space of designs, it is completely automated.

Another aspect of our technique is the generality of the rules. Consider the case of a completely new design where our current set of rules are not sufficient to prove the equivalence. Following the proof trace, it is fairly straightforward to add the required rules for a 4-bit or 8-bit version of the multiplier. However, if the multiplier RTLs are modular, then these rules are general enough that the harder cases of 32-bit and 64-bit multiplier equivalence proofs are now completely automated.

3.5.3 Results

We present the experimental results that we have obtained from our tool. We produce three sets of results, on a radix 3 Booth multiplier, on a Wallace Tree multiplier and on a Dadda Tree multiplier. The Booth multiplier is an array-based multiplier, whereas the Wallace multiplier has a tree of carry save adders and a single carry lookahead adder as the last stage. The Dadda Tree multiplier uses the more regular redundant binary addition trees[228] instead of a tree of CSAs. We show the time taken by the tool for increasing sizes of these multipliers.

We have tried to compare our tool to state-of-the-art equivalence checkers. Since the equivalence checkers are most efficient when comparing two gate level designs, we provided gate level implementations of the Booth and Wallace Tree designs as inputs. Although our tool works at the RT level, we have compared the numbers obtained from the gate level verification by the equivalence checkers with our tool output, in order to provide a basis for comparison. It is seen from Figure 3.9(a), Figure 3.9(b) and Figure 3.9(c) that the verification of $8X8$ multipliers are performed by both Commercial Equivalence

Checker 1 and Commercial Equivalence Checker 2 in time comparable to our tool. However, in the case of 16×16 multipliers, both the equivalence checkers do not run to completion. Our tool, in comparison, verifies the design in 24 seconds. It can also be seen that as the sizes increase, the time taken by our tool scales linearly with the size of the design.

The *reduce()* function uses a database of rewrite rules to prove equivalence. Figure 3.10 illustrates a broad classification of the multiplier rewrite rules. We have classified the rules into those involving Boolean operators, add/subtract operators and shift operators. These rules are mostly generic in nature, and will form a part of the rule database for all multipliers. However, the rules classified into multiplier specific rules are more specific to the design of the multiplier being verified. As mentioned earlier, the set of rewrite rules is not exhaustive. However, for the widely used multiplier designs, the necessary set of rewrite rules exists in the database, thereby making the rewriting very efficient.

We mentioned in Subsection 3.3.3 that the *reduce()* function proves term equivalence using proof iterations. Figure 3.11 shows the number of proof iterations that the tool required to prove the equivalence of the 64×64 multipliers at two sample compare points. The proof iterations for the Booth multiplier at (sample) compare points 3 and 21 and the Wallace Tree multiplier at compare points 3 and 7 have been illustrated. We observe that the proof iterations do not increase significantly as the proof progresses, *i.e* with increasing comparison points. We also observe that the number of compare points for the Wallace Tree are lesser than the Booth. This is because of the tree of carry save adders in the Wallace Tree design which delays assignment to the final output bits. Therefore, the terms are larger and the effect of this is seen in the increased number of proof iterations for the Wallace Tree than for the Booth multiplier.

In order to assist the Commercial Equivalence Checker 1 to compare RTL designs, we tried providing comparison points in the multiplier designs. These intermediary cutpoints were the partial products obtained in the two multipliers. The results of this experiment are displayed in the Figure 3.12. The

Commercial Equivalence Checker runs to completion when assisted manually with comparison points for the 16×16 case. However, it fails to run to completion, even when assisted by these comparison points, when comparing two 32×32 bit or higher order multipliers. It may be noted that we have provided manual assistance with respect to the comparison points to the equivalence checkers, as opposed to our tool that generates these comparison points automatically. Our tool is effective in verifying multiplier designs that are modifications (usually for optimization) to standard multipliers like Booth, Wallace Tree and Array multipliers. We used the tool for verifying the Verilog implementation of BISMUL [249], a complicated, modified Booth multiplier. In this case, a Booth multiplier verified by our technique was used as the golden design, and the BISMUL was the target design to be verified. Our tool caught a bug in the Verilog code, that appeared while the tool tried to calculate the partial products after the first matching point. The expressions that the observed output (product) variable (P) was rewritten into, could not be proved equal at the next matching point by Vprove. The rule correspondence that the tool had established, as well as the previous matching point, provided an error trace.

3.6 Conclusions

We have presented a new approach that uses stepwise refinement of TRSs for formally verifying arithmetic circuit designs. Equivalence checkers that use BDD-based algorithms [175] cannot handle large sizes of multipliers. Our tool manages to gracefully scale to large, complex multipliers.

We provide a brief intuition as to why we demonstrate better performance. The first reason is due to our strategic decomposition of the equivalence checking state space. The process of computing comparison points once, before starting the verification process, by statically analyzing the RTL descriptions helps in decomposing the intractable problem. These comparison points help us obtain smaller

equivalence checking problems, that can be verified using term rewriting, or other equivalence checking engines. Since this process is done statically, it does not depend on the size of the design.

Another reason is because we represent circuits at a higher *term* level as opposed to the Boolean level representation used by the BDD based techniques. The term representation we use is intuitive and easy. It is also more natural, since the terms encapsulate the structural as well as the functional details of the circuit in a modular fashion. The manipulation of terms is also more efficient. Also, while comparing two TRSs, the problem of comparison can be reduced to a smaller problem, since two terms being compared need not be in their normal (canonical) form. However, BDDs are a canonical representation, and their comparison cannot be easily decomposed. Our decomposition of two TRSs to pairs of comparable terms is also a significant reason for the efficiency of the technique.

The disadvantage in term rewriting is that the Vprove part of the tool, that implements the *reduce()* function introduced in Section 3.3, is incomplete. The *reduce()* function uses a database of rules to simplify the expressions it is comparing. This database of rules may require additional rules to simplify new expressions. However, this incompleteness is traded for the efficiency of the tool, and in practice it can deal with many classes of large multipliers which were hitherto intractable with automated tools. We have incorporated a large number of rules that were needed to simplify the expressions that we encountered in the circuits we have targeted. In its current state, therefore, Vprove is very efficient for practical designs.

For arithmetic circuit verification, some techniques [20, 37, 54, 55, 108, 248] are more effective than other model checking techniques. However, our technique achieves significantly more, since we *automatically compute comparison points*. Binary Moment Diagrams (BMDs) have been shown to verify $256bit$ wide multipliers in reasonable time. However, BMDs are constructed by partitioning the circuit into components, each component having simple word-level specifications. This custom-crafting is not a

feasible option for a generic design. Also, practical design optimizations tend to break such elegant component constructions. Our technique, on the other hand, does not need to construct artifacts according to the individual designs, and is far more generic in scope. We are particularly able to handle complex design optimizations, as shown in Subsection 3.5.2. Verification times for BMDs are a quadratic function of the size of the multiplier, as opposed to our technique, where the times scale linearly with size. Additionally, BMDs are also prone to variable ordering issues(albeit, to a lesser extent than BDDs) that require significant manual intervention. When the designs being checked are not found equivalent, the BMD algorithm may not terminate. In contrast, our tool aborts and provides an error trace for the bug instantly. Since the equivalence checking is decomposed into comparison points, the exact location of the error can easily be detected at a failing comparison point.

Our technique is similar in spirit to a directed theorem proving approach. However, our technique requires much less user expertise and ingenuity than theorem provers [72, 73, 98, 111, 139, 142, 170, 202, 209, 212]. Our tool is a dedicated arithmetic circuit checker, and can be interfaced with equivalence checkers for arithmetic circuit verification. Another possibility is to integrate our tool with the theorem prover ACL2 [144], so that we can leverage the existing RTL library in ACL2 [75, 169] to add rules to Vprove in a sound manner.

Our technique is a step toward verification of two generic arithmetic circuits. We have managed to verify a large number of arithmetic circuits using our technique, like adders, shifters, and comparators. This technique can tackle a large part of the multiplier space, and many of the multipliers currently in use. We are also working on applying the technique to SRT division circuits and floating point arithmetic circuits.

Booth Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	16s	12s	9s
$8b \times 8b$	19s	20s	16s
$16b \times 16b$	24s	not completed	not completed
$32b \times 32b$	37s	not completed	not completed
$64b \times 64b$	53s	-	-
$128b \times 128b$	93s	-	-

(a) Booth Multiplier

Wallace Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	14s	10s	9s
$8b \times 8b$	18s	18s	16s
$16b \times 16b$	25s	not completed	not completed
$32b \times 32b$	40s	not completed	not completed
$64b \times 64b$	60s	-	-

(b) Wallace Tree Multiplier

Dadda Tree Multiplier	Verifire	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	13s	11s	8s
$8b \times 8b$	17s	19s	17s
$16b \times 16b$	29s	not completed	not completed
$32b \times 32b$	51s	not completed	not completed
$64b \times 64b$	83s	-	-

(c) Dadda Tree Multiplier

Figure 3.9. Comparison of execution times of Verifire against two commercial equivalence checkers for Booth, Wallace Tree and Dadda Tree multipliers of varying sizes. In each case the golden model was a shift and add multiplier of the corresponding size.

Rule class	Number of rules	Example
Boolean	32	`(x | (y & z))` → `((x | y) & (x | z))`
Add/Subtract	44	`(x + (y - z))` → `((x + y) - z)`
Shift	16	`((x << 1) << 1)` → `(x << 2)`
Multiplier Specific	9	`((x << 1) - x)` → `x`
Total	101	

Figure 3.10. Distribution of rewrite rules for multipliers used by the *reduce()* function of Vprove.

Multiplier	Compare point	Number of rules	Number of proof iterations
Booth	3	107	192
Booth	21	107	212
Wallace Tree	3	107	347
Wallace Tree	7	107	291
Dadda Tree	3	107	462
Dadda Tree	7	107	341

Figure 3.11. Number of proof iterations done by *reduce()* to prove equivalence at the given compare points. The numbers correspond to the 64×64 Booth, Wallace Tree, and Dadda Tree multiplier designs.

Multiplier	Verifire (Booth)	Commercial Tool (Booth)	Verifire (Wallace)	Commercial Tool (Wallace)	Verifire (Dadda)	Commercial Tool (Dadda)
$4b \times 4b$	16s	12s	14s	10s	13s	8s
$8b \times 8b$	19s	20s	18s	20s	17s	17s
$16b \times 16b$	24s	1942s	25s	972s	29s	not completed
$32b \times 32b$	37s	not completed	40s	not completed	51s	not completed
$64b \times 64b$	53s	-	60s	-	83s	-

Figure 3.12. Comparison of execution times of Verifire against one commercial equivalence checker assisted by manual comparison points. Results are shown for Booth, Wallace, and Dadda tree multipliers.

Chapter 4

Sequential Equivalence Checking at System Level and RTL

4.1 Introduction

System-on-a-chip (SoC) designs contain unprecedented levels of functional and structural complexity in a single system, making their verification a daunting challenge to known verification methodologies. Inordinate amounts of time and effort are spent in the SoC industry, validating a chip for functional and timing requirements. Although simulation based verification is the most widely used technique for validating SoCs, the degree of confidence in these simulations is low for these highly complex, monolithic designs. Formal verification is desirable due to its high quality assurance, but the known techniques of hardware or software verification are not equipped to handle the size, or the heterogeneity of SoCs. Futuristic verification research then, involves new formal methods that are capable of scaling to the SoC domain.

The problem of checking sequential equivalence of a system level model (SLM) with respect to its implementation in Register Transfer Level (RTL) is a relatively novel domain. An equivalence check at this level, or even at the RTL to RTL level is desirable due to a number of reasons [198]. Optimizations for power, speed, area or other design parameters is usually done during this stage of design. These

optimizations could range from shifting logic between flip-flops, or using latches in place of flip-flops in certain portions, or changing the algorithm of a certain portion of the design to a faster one. In all these cases, an equivalence check at this level would enhance the confidence in the optimized implementation, as well as be more economically viable than detecting bugs at a later stage in the design cycle.

We present a sequential equivalence checking technique to verify system level design descriptions against their implementations in RTL using a three-pronged solution based on high level static analysis.

Domain aware analysis

Our technique involves the efficient decomposition of the equivalence checking problem, in order to make it more tractable. We present an automatic technique to compute high level *sequential compare points*, to compare variables of interest (observables) in the candidate design descriptions. Our compare points are defined as co-ordinates on the space-time axis of the design, denoted by their relative position with respect to the time domain (clock cycles), and their position in the space domain (data variables). This aligns with the sequential behavior of the designs being compared, and provides an easy, intuitive abstraction of the equivalence checking problem space. We start the two design state machines at the same initial (or reset) state, and step the machines through every cycle, until we reach a sequential compare point.

High level symbolic simulation

At the sequential compare points, we construct symbolic expressions for the observables that encapsulate the sequential behavior of the designs, until the cycle of comparison. At each sequential compare point, we prove the equivalence of the two state machines. using a lower (Boolean) level engine. The principal gain in our technique is that we are leveraging the expressiveness and information available to us at the register transfer and system levels. Although significant amount of research has been done on compare points for gate level equivalence checkers, these algorithms and heuristics are limited by their domain.

On the other hand, since we operate at the higher, source code level, our sequential compare points are more intuitive and easier to detect. Also, they capture the notion of design progression through time, which is useful in meaningful decomposition of the equivalence checking state space.

Decision procedure

We use a Boolean satisfiability (SAT) solver as the decision procedure for checking the equivalence of the expressions built by high level symbolic simulation.

The principal contributions of this work are the following.

- We present a sound sequential equivalence checking method between system level design descriptions and their RTL implementations
- We present an automatic decomposition technique for splitting the equivalence checking problem space, by introducing a notion of sequential compare points that exactly model the sequential behavior of designs.
- We leverage the expressive power and relative simplicity of high level descriptions in our equivalence checking, by reasoning entirely at that level.
- Our technique can statically be used to analyze and decompose the source code, in order to assist Boolean level engines to overcome capacity issues.
- We demonstrate the effectiveness of our technique by checking the equivalence of the implementation of a real SoC component against its specification.

The outline of the chapter is as follows. Section 4.2 provides an overview of the related work in this, and allied areas. Section 4.3 details the technique, the algorithm for automatic decomposition, and

explains it with a simple example. Subsection 4.3.3 gives the proof of correctness of our technique. In Section 4.4, a case-study of the Viterbi decoder is presented. Subsection 4.4.1 and Subsection 4.4.2 describe the two implementations of the Viterbi decoder and the verification process. The result of our experiments are presented in Subsection 4.4.3. Section 4.5 contains a brief discussion and concluding remarks.

4.2 Related Work

We provide a brief background that explores the related work in the entire spectrum of topics covered by this work.

Formal verification, especially equivalence checking, has achieved considerable success in the context of hardware. Combinational equivalence checking checks two acyclic, gate-level circuits. Combinational equivalence checkers can also be used to check equivalence of two sequential designs, provided the state encodings of the two designs are the same. Although this technique has widespread use in many commercial tools, the real challenge of sequential verification is in verifying two designs with different state encodings. Sequential satisfiability engines [166, 200] and sequential ATPG engines [9, 127] solve this problem to a large extent by unrolling the circuit until a given time frame. Considerable research has been done to find compare points for latch mapping [15, 44, 235]. However, these techniques operate at the gate level, where they reason in the Boolean domain.

Fewer attempts have been made to apply sequential equivalence checking to the behavioral RTL descriptions of designs. In [219] a methodology for checking the combinational equivalence between C and RTL is described. The C source code is converted to a Hardware Description Language (HDL) and commercial RTL to RTL equivalence checkers are used thereafter. The C code is very similar to the

RTL, in order for the translation to be achieved, which might not be a scalable solution.

Clarke and Kroenig [63, 153] proposed a solution with CBMC, a C-based bounded model checking engine that takes a C program and a Verilog implementation. The two programs are unwound together, and converted into a Boolean satisfiability checking problem. The Verilog code is converted to Boolean formulas by a synthesis-like procedure, and an innovative technique is described to convert the C-code into Boolean formulas, including pointers and nested loops. However, the capacity of CBMC is limited by space and time considerations. This is due to the fact that the reasoning done by this tool is entirely in the Boolean domain. On the other hand, our technique reasons at the system and register transfer level, splitting the equivalence checking problem into smaller problems that can be handled by the lower level engines. This static analysis of the source code, before running the problem through Boolean level engines, is the principal contribution of our technique.

Another approach to equivalence checking between C descriptions, that could be extensible to C vs RTL descriptions, is described in [174]. This approach detects and extracts the textual differences in the two target programs, and then does a dependence analysis using program slicing, to check for the actual differences in the two programs. It then symbolically simulates this difference and reports the equivalence checking results. This technique, however, is most effective when the two target programs being compared are very similar to each other, in function as well as structure. Since this process uses syntactic information entirely, the similarity of the target descriptions is very essential to its application. Our technique does a semantic comparison of the two target programs, with respect to their functionality, and is therefore wider in its scope.

A few commercial tool vendors [1] also aim at solving the sequential equivalence checking problem between SLM and RTL. However, this area still presents a major opportunity for further research.

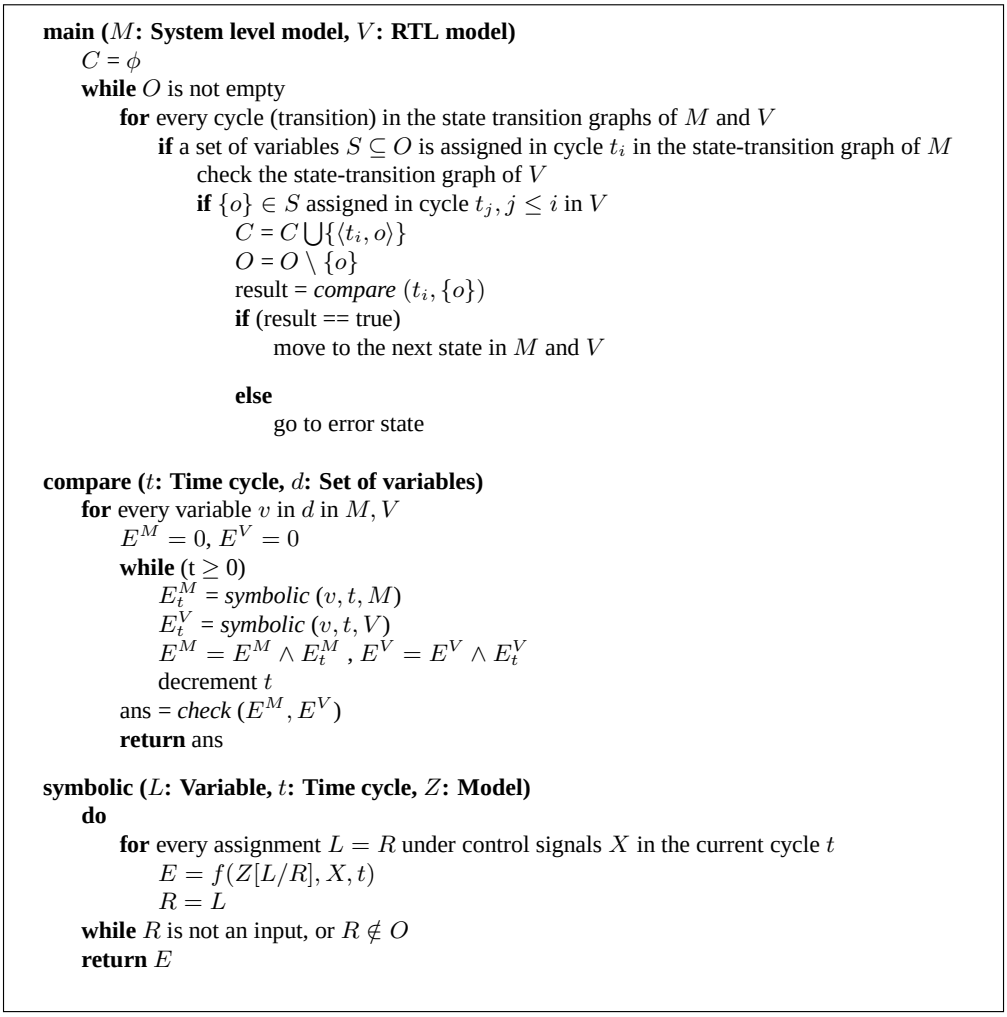


Figure 4.1. Algorithm for proving equivalence between a C-like system and its RTL implementation

In previous work, [227] we presented a technique for RTL to RTL equivalence checking of complex combinational circuits, including multipliers. We extend our technique to sequential equivalence checking, and address the problem in the realm of SLM vs RTL.

4.3 Technique for System level vs RTL equivalence checking

We present a technique for equivalence checking of two high level design descriptions. Our technique involves the automatic decomposition of the equivalence checking state space, from the source code of two candidate designs. We introduce the notion of sequential compare points, that encapsulate the sequential behavior of designs, with respect to time as well as data. In the rest of the paper, we will refer to sequential compare points, simply as compare points.

4.3.1 Algorithm

An algorithm for our technique is presented in Figure 4.1.

Let M and V denote the (C-like) specification and the RTL implementation respectively. The Kripke structure or the state-transition graphs for these systems at the source code level is constructed. A system state consists of symbolic values of all state variables in the system, and a transition corresponds to progression of time, with respect to the explicit clock in the system. The signals that are interesting for observation are a part of the list of observables, O. Typically, this list is obtained from the primary outputs, or a block diagram of the C and the RTL designs. This list is the same for both the systems, as it is assumed that a name mapping is provided for all the primary inputs and observable signals in the designs. The systems are assumed to start in the same initial states. C denotes the list of all compare

points. A compare point has a two-tuple description, one for the time cycle, and another for the set of variables that are being compared. This list is empty initially.

The state transition graphs of the two systems are traversed step-by-step. This can be thought of as stepping both the systems in time. After every transition (cycle), the two systems are checked to see if any of the observable variables are assigned. If a variable from the list of observables is assigned in one of the systems (say M), the other system (say V) is checked to see if the same variable has been assigned in the current, or some previous cycle. If the variable was assigned in V before M, the two systems are compared at the current cycle, *i.e* the current cycle becomes a compare point. If, however, the variable has not yet been assigned in V, the two systems are transitioned until the next observable is assigned in either.

The *compare()* function compares the observables at a given cycle. Since there may be more than one observable that is being compared at a cycle, this function takes a set of variables. For each of these variables, a symbolic expression is computed at the "current" time cycle, t. The symbolic expressions computed in every previous cycle, until $t = 0$ are iteratively concatenated in both the systems, to obtain E^M and E^V. The two expressions E^M and E^V are now checked for equivalence using a SAT solver. The *check()* function corresponds to the SAT solver call in our algorithm. It may be noted, however, that other equation solving engines may also be employed to check the equivalence of expressions at this stage.

The *symbolic()* function computes the symbolic expression at a given cycle. For every assignment to a given variable, it substitutes the right hand side of the definition for the variable (denoted by $Z[L/R]$ for any model Z) along with the control signal information X, required for the substitution. X is a Boolean expression that represents the constraints on the control signals. The substitutions $M[L/R], V[L/R]$ are valid when X is true. The symbolic expression also identifies the cycle t in which it holds true. This

expression is computed by $f()$ in the algorithm.

If during this substitution, a primary input or a previously "observed" observable is reached, the substitution stops. This is valid because the two systems have been proved equivalent with respect to this observable in a previous compare point. From that cycle onwards, the two systems are always equivalent with respect to the observables at that compare point. In a sequential design, the data from the previous cycle progressively gets used in future time cycles. The symbolic expression of the observables at a given compare point remains the same for all future compare points. Therefore, these observables need not be proved equivalent, at every reference to the corresponding compare point.

If they are found equivalent, the proof proceeds. The comparison process is repeated, until all observables have been "observed". If they are not found equivalent, an error trace is generated at that compare point. This is very useful, since the equivalence proof can be assumed to hold until the compare point where it fails.

4.3.2 Example

We show the process of building symbolic expressions for comparison using an example system in RTL. This is trivially extensible to System C.

Figure 4.2 shows a code segment written in Verilog RTL.

The `always` block shows a concurrently executing process. The parentheses of the always block list the signals on which the process is dependent. In this case, the `always` block will be executed, whenever there is a change in the the clock, or the `clk` signal, *i.e* at every cycle. The output of the given Verilog design is `result`, a 2 bit variable.

Let `result` be the observable. The state-transition graph for the Verilog design V, is shown in

```
always @(clk)
  begin
   if (reset) {
           a = 0;
           b = 0;
   } else {
        if (sel) {
              a = 1;
              b = 0;
        } else {
              a = 0;
              b = 1;
        }
   }
   result[0] = a;
   result[1] = b;
  end
```

Figure 4.2. Example Verilog RTL code.

Figure 4.3. The variables whose values are updated, as well as the corresponding control variables in each state are shown. From Figure 4.1, we construct the symbolic expression E for the given design. From the state-transition graph, it can be seen that `result` is assigned to, in the second clock cycle. Therefore, t is the second clock cycle. For the assignments `result[0] = a` and `result[1] = b`, $E_t = f(V[result[1]/b, result[0]/a], (reset = 0), t)$. Therefore, $E_t = t \wedge (reset = 0) \wedge result[1]/b \wedge result[0]/a$. In the second iteration, $t = t - 1$. $E_{t-1} = ((t-1) \wedge (reset = 0) \wedge (sel = 0) \wedge (a/0) \wedge (b/1)) \vee ((t-1) \wedge (reset = 0) \wedge (sel = 1) \wedge (a/1) \wedge (b/0))$. In the third iteration, $t = t - 2$. $E_{t-2} = (t-2) \wedge (reset = 1) \wedge (a = 0) \wedge (b = 0)$. E is obtained by a logical AND operation of E_t, E_{t-1} and E_{t-2}.

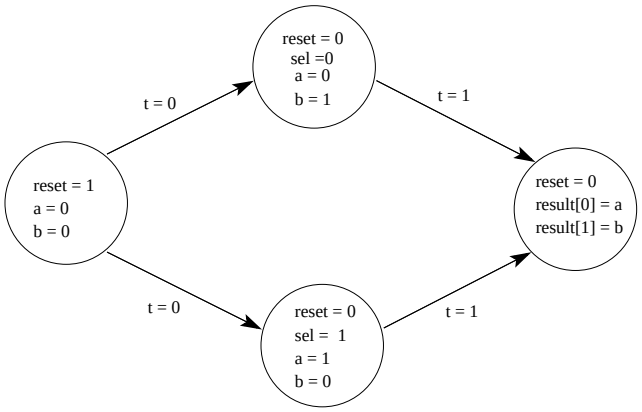


Figure 4.3. State-transition graph of example Verilog RTL. The transitions correspond to consecutive clock cycles. The states are indicated by the symbolic values of variables. Only relevant variables are shown in every state.

4.3.3 Theoretical Justification

We present here, a theoretical basis to justify our technique.

Let M be the design specification system (model). Let V be its implementation. Let $PI(X)$ and $PO(X)$ denote the primary inputs and primary outputs of system X, such that $PI(M) = PI(V)$ and $PO(M) = PO(V)$. We assume that a signal name mapping is provided between the two systems. Let $\sigma_X(s,t)$ be the function in a system X that takes in a signal s at a point t in time, and returns the symbolic expression for s in terms of $PI(X)$ over all times until t. We define the simulation relation, $\sim_{p,t}$ where p is a set of signals, as the function between two systems X and Y, such that $\forall i \in p, \sigma_X(i,t) \equiv \sigma_Y(i,t)$. Our notion of equivalence is to show that $V \sim_{PO} M$.

We define a compare point $C = (t,d)$ as a co-ordinate on the space-time axis of the design, denoted by its relative position with respect to the time domain t, and its position in the space or data domain d.

In the context of designs, t would be in terms of cycles, and d would be a set of observable signals.

Lemma 4.3.1. *At a given t, $\forall i \in d$, where $C = (t, d), V \sim_{i,t} M \Rightarrow V \sim_C M$.*

Proof outline: For any observable signal i at a given time step, $q = \sigma_V(i, t)$ is equal to the expression obtained by iteratively expanding i for every time step until t, such that the symbolic values are obtained in terms of the primary inputs, $PI(V)$. $r = \sigma_M(i, t)$ can also be similarly interpreted. In order to prove that $q \equiv r$, we use a SAT solver or another engine whose functionality we assume is correct. Once $q \equiv r$ is proved, it implies that for all input values, at cycle t, the two systems will have the same value for i. If this result is proved for every $i \in d$, since t is a given constant, we can conclude that a simulation relation holds between the two systems.

Theorem 4.3.2. *Let the two systems M and V be described with $PI(M) = PI(V)$ and $PO(M) = PO(V) = P(O)$. Let n be the longest cycle length (time step) taken to obtain all primary outputs in both systems. Let M and V be compared at at every compare point $C = (t, d)$, such that $0 \leq t \leq n$. Then, $\forall C, V \sim_C M \Rightarrow V \sim_{PO} M$.*

Proof outline: The proof follows from induction, where the base case is at time $t = 0$, when the systems start from the same initial state. The induction hypothesis is relieved by using Lemma 4.3.1 and substituting equals for equals in the entire symbolic expression obtained. If all the primary outputs are generated by cycle n, at $C = (n, PO)$, the desired simulation relation will hold between the two systems.

4.4 Equivalence Checking of System C vs RTL of Viterbi Decoder

We perform our experiments on a Viterbi decoder, that is a part of the Digital Radio Mondiale (DRM), implemented in System C. Since the Viterbi decoder module (embedded in the MLC decoder) took an inordinately long number of simulation cycles, the DRM SoC design was partitioned to implement the Viterbi decoder in hardware. This hardware accelerator was implemented in Verilog RTL, from the initial System C description of the Viterbi module. More details on this process can be obtained from [232].

Optimizations for speed, like pipelining, are typical applications where an equivalence checking between the system level design and the RTL are desired. Our experiments use such optimized implementations to show the efficacy of our technique.

The System C specification of the Viterbi decoder is a very basic model, that implements the Viterbi decoding algorithm, but has no optimizations for speed, area or power (Figure 4.4). The first RTL design we compared against, is a pipelined implementation of the Viterbi decoder, optimized for speed. The second implementation is optimized for area. We show the results of doing equivalence checking using our technique on these Verilog designs, with respect to the specification in System C.

Figure 4.4 shows the basic block diagram of a Viterbi decoder [244]. There are two major stages to the functionality of the Viterbi decoder. One is collecting the inputs depending on the Puncture Pattern and storing them in an buffer (FF Buffer). The other stage is the Trellis computation. The next state values of the Trellis matrix are computed by a function (Butterfly network) of current state values of the Trellis matrix and the inputs stored in the FF Buffer.

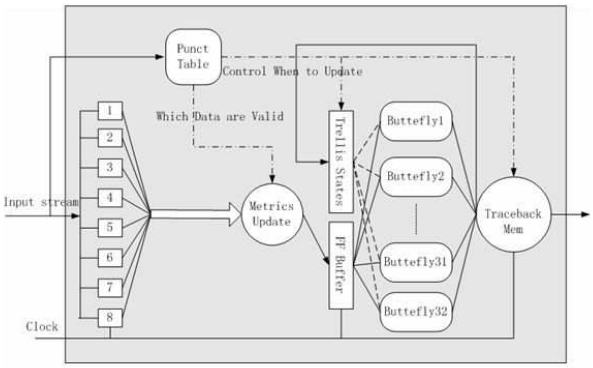


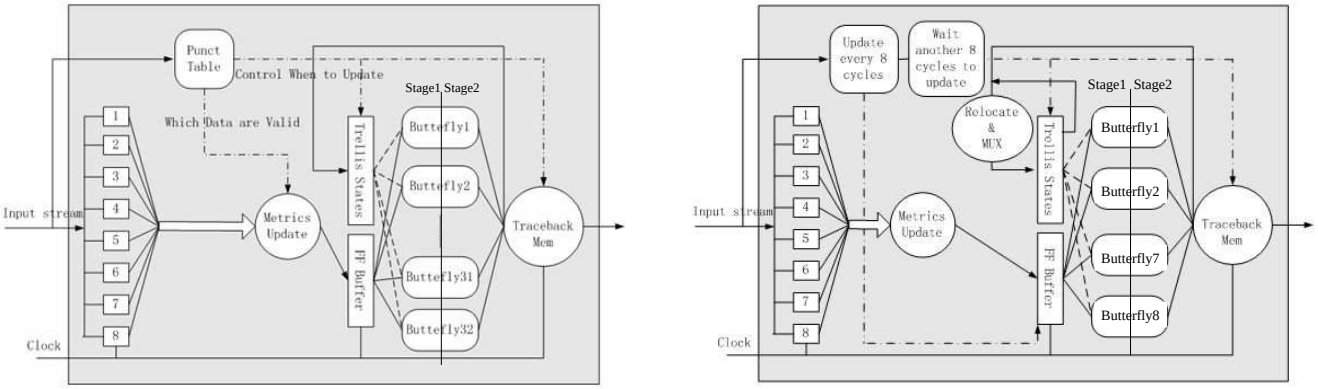
Figure 4.4. Viterbi decoder- System C design specification

4.4.1 Equivalence Checking of a Pipelined Viterbi Design

We started with a System C description, as well as the pipelined Verilog RTL implementation of the Viterbi decoder design. Figure 4.5(a) shows the block diagram of the pipelined implementation. From the block diagram, we arrive at the following observables in the experiment.

- 8 FIFO entries, each 32-bits wide: FF[7:0][31:0]
- 64 Trellis Matrix entries, each 32-bits wide: TM[63:0][31:0]
- 2 entries in the MatDec, each 32-bits wide: MD[1:0][31:0]
- Decoded output, 32-bits wide: Out[31:0]

The signal (variable) mapping between the two designs is provided. For the sake of readability, we denote the observables in the System C design with a subscript s, and the observables in the Verilog design with a subscript v. We outline the proof methodology using our technique. We start both the



(a) Viterbi Decoder with 32 2-stage Butterfly blocks **Design 1**

(b) Viterbi Decoder with 8 2-stage Butterfly blocks **Design 2**

Figure 4.5. Viterbi Decoder block diagram. The decoder on the left is a pipelined Viterbi decoder with a 2-stage Butterfly, with 32 parallel butterfly blocks. The decoder on the right is further optimized for area with only 8 parallel butterfly blocks. The area-optimized design will run 4 times slower on the Trellis computation.

designs at the reset state initially. We step the two designs in tandem. From the state-transition graph of the System C design, we observe that the output is computed at every cycle. From the state-transition graph of the Verilog design, however, we observe that the output is computed in the 10^{th} cycle after the reset state. In accordance with our algorithm in Figure 4.1, we need to step the Verilog design more than the System C specification, to arrive at compare points. Figure 4.4 is a pictorial representation of the decomposition of the equivalence checking proof, on the basis of compare points. The horizontal axis represents the data (observables), and the vertical axis shows the number of systems being compared (in our case, two). Time is represented along the axis normal to the plane of the paper.

The first set of observables FF[7:0][31:0] is available after 8 cycles, at the output of the FF Buffer. The first compare point, is therefore $C_1 = (t = 8, d = FF[7:0][31:0])$.

For each entry i in the FIFO buffer, the FIFO variables are FF_s[i][31:0] and FF_v[i][31:0]. We call the *compare()* and *symbolic()* functions at the compare point, and obtain the expressions for the FF variables.

In both the designs, the FF Buffer gets updated by the function GetMetricSet(). Therefore, the symbolic expressions correspond to an expansion using this function. The two symbolic expressions for FF_s[i][31:0] and FF_v[i][31:0] are checked using a SAT solver. This procedure is repeated 8 times, for every entry in the FF Buffer, since each of them has a unique symbolic expression.

The next comparison point is obtained by stepping the two state machines of the designs after the 8^{th} cycle. Although the System C assigns to an observable every cycle, the Verilog design assigns to the next observable at the 10^{th} cycle. The next $v \in d$ is the Trellis Matrix, TM[63:0][31:0]. All the entries in this 64×32 matrix need to be checked, since the entire table is updated every 10^{th} cycle. The values of the MatDec decision table, MD[1:0][31:0] is also updated in this cycle, as is the decoded output, Out[31:0]. The intermediate variable every 9^{th} cycle, btm which is not an observable is shown in lower

case in Figure 4.4.

The second compare point, is therefore, $C_2 = (t = 10, d = TM[63:0][31:0], MD[1:0][31:0], Out[31:0])$.

The Trellis Matrix table gets its values from the 32 butterfly blocks in the design, each of which output 2 entries. The symbolic expression from the RTL, therefore, is a function of the butterfly blocks. For every 2 entries in the Trellis Matrix, the corresponding symbolic expression can be obtained from the butterfly. For instance,

TM_v[0], TM_v[1] = Butterfly(TM_v[0], TM_v[2], FF_v[0][31:0], FF_v[7][31:0])

Similarly, in the System C design,

TM_s[0], TM_s[1] = Butterfly(TM_s[0], TM_s[2], FF_s[0][31:0], FF_s[7][31:0])

Since FF_v[0] = FF_s[0] from a previous comparison point C_1, the symbolic expression for these signals are not expanded any further. The symbolic expressions for TM_v[0], TM_v[1] and TM_s[0], TM_s[1] are checked for equivalence by the SAT solver. This procedure is repeated 32 times, for every pair of entries in the Trellis Metric that need to be checked.

The other observables MD[1:0][31:0] and Out[31:0] are similarly checked for equivalence. The proof of Out[31:0] is not shown in the figure. The results of the *check()* function are discussed in the next section.

4.4.2 Equivalence Checking of a Pipelined Viterbi Design Optimized for Area

We used our technique to perform equivalence checking of a pipelined Viterbi design, that is further optimized for area. In this design, the 32 butterfly units are split into 4 stages, each stage having 8 butterfly units. Figure 4.7 shows the decomposition of the proof using compare points.

Figure 4.7a shows the same proof progression as discussed in Subsection 4.4.1 with respect to the outputs of the FIFO buffers. As in the previous example, the first compare point is $C_1 = (t = 8, d = FF[7:0][31:0])$. Thereafter, the two state machines are stepped in tandem. The next observables, namely the Trellis Matrix and the MatDec decision table values are computed at the end of the $10th$ cycle. However, since the butterfly unit has been divided into 4 stages, only 16 values of the Trellis Matrix and 8 values of the MatDec table are obtained at the end of this cycle. The second compare point, therefore, is $C_2 = (t = 10, d = TM[15:0][31:0], MD[1:0][7:0])$ Similarly, the other compare points are

$C_3 = (t = 12, d = TM[31:16][31:0], MD[1:0][15:8])$,

$C_4 = (t = 14, d = TM[32:47][31:0], MD[1:0][16:23])$ and

$C_5 = (t = 16, d = TM[47:63][31:0], MD[1:0][23:31], Out[31:0])$. The decoded output gets computed at the end of the 16^{th} cycle.

The symbolic expressions for TM_s[63:0][31:0] and TM_v[63:0][31:0] are similar to those described in the previous proof subsection. Similarly, the values of the other observables can be symbolically computed and checked with a SAT solver.

It hold be noted that this proof has more compare points than the proof shown in Figure 4.4 and also requires a sequential progress over more time cycles.

4.4.3 Experimental Results

We use zChaff [96] as the SAT solver to implement the *check()* function. In order to pass the equivalence checking through zChaff, we had to model the symbolic expressions as a Boolean satisfiability problem. We used the XNOR operation to combine the two target symbolic expressions. In order to provide a glimpse into the complexity and size of our design, we present some relevant statistics in Figure 4.8. Table A gives a breakdown of number of clauses in the CNF formula for various blocks. PLUS and LESSTHAN were two primary functions used to synthesize the RTL into symbolic boolean expressions. We can observe the benefits of our technique of splitting monolithic equivalence functions into smaller functions. Using our decomposition technique, we created 32 independent CNF formulas, that were input to zChaff. Each of these formulas had 59136 clauses and 128 variables. Without this decomposition, the monolithic Trellis computation would generate a CNF with nearly 1.9 million clauses. An interesting observation is, due to our decomposition methodology, the size of the CNF for the pipelined version is exactly the same as the non-pipelined design. This shows that the equivalence checking problem can be greatly reduced using a high level decomposition, in order to make it easier for lower level engines. These observations are captured in tables Table B and Table C.

4.5 Summary

We have shown a novel technique for sequential equivalence checking of a system level specification and its implementation in RTL. Our technique decomposes the equivalence checking problem using automatically computed compare points. We demonstrate the efficiency of our technique using a non-trivial example.In principle, the technique can be also be applied to RTL to RTL equivalence checking, since it is effective for source to source checking at the higher level.

We show our technique using System C as our system level modeling language, due to its recent IEEE standardization. Our technique, however, can be applied to other system level languages also. In its current form, this technique cannot handle pointers, nested loops or other software specific constructs. In future, we plan to extend this work to handle a larger subset of the C-like description language.

Since this work primarily seeks to decompose the problem into more tractable sub-problems, it might be usable in conjunction with existing tools and techniques. The information content at the source code level can be exploited to assist existing technologies to tackle the SoC verification problem. This work motivates the necessity to reason at the higher levels in the design cycle, in order to work toward scalable verification solutions.

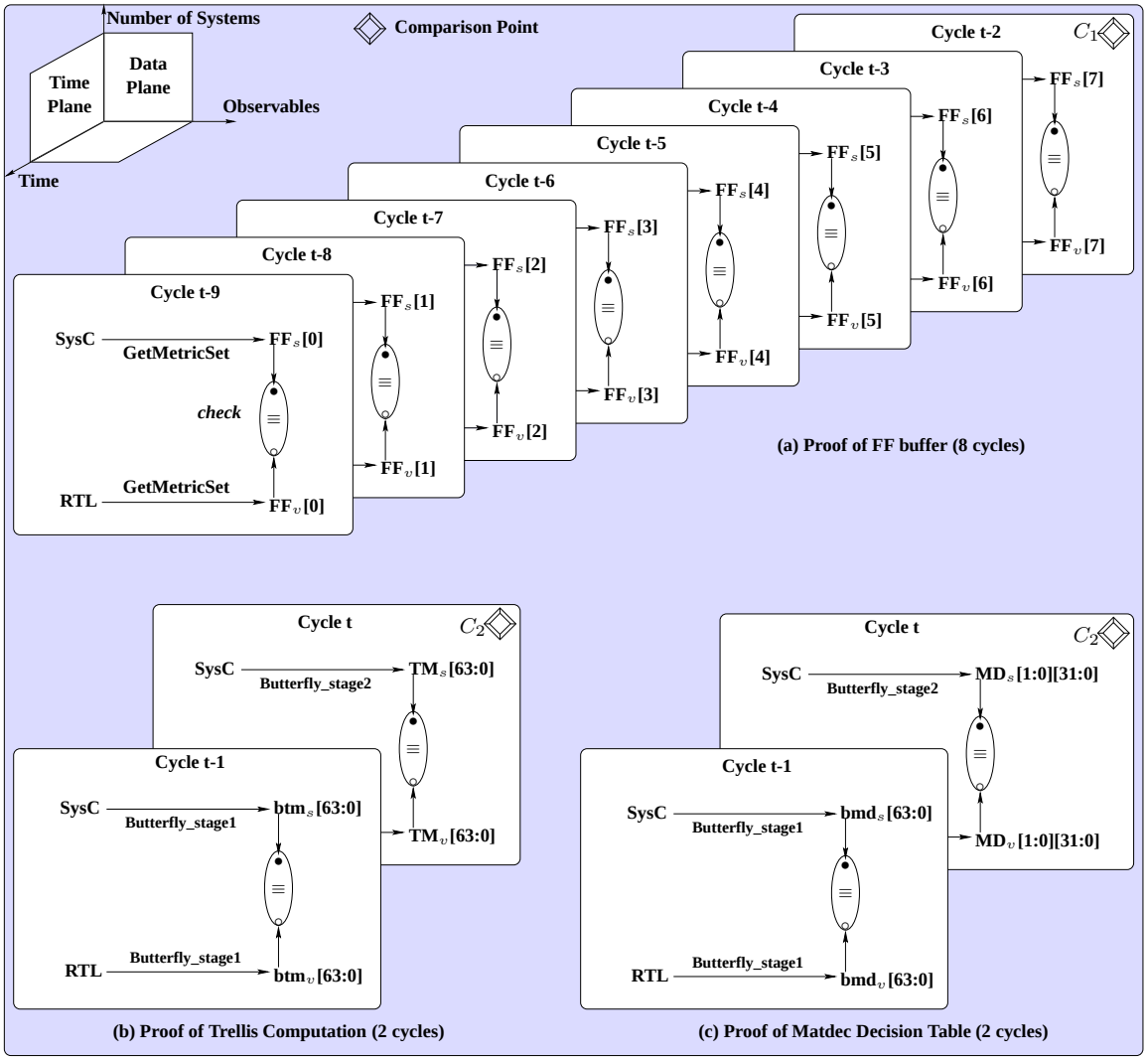


Figure 4.6. Proof of sequential equivalence checking of pipelined Verilog Viterbi design against System C design

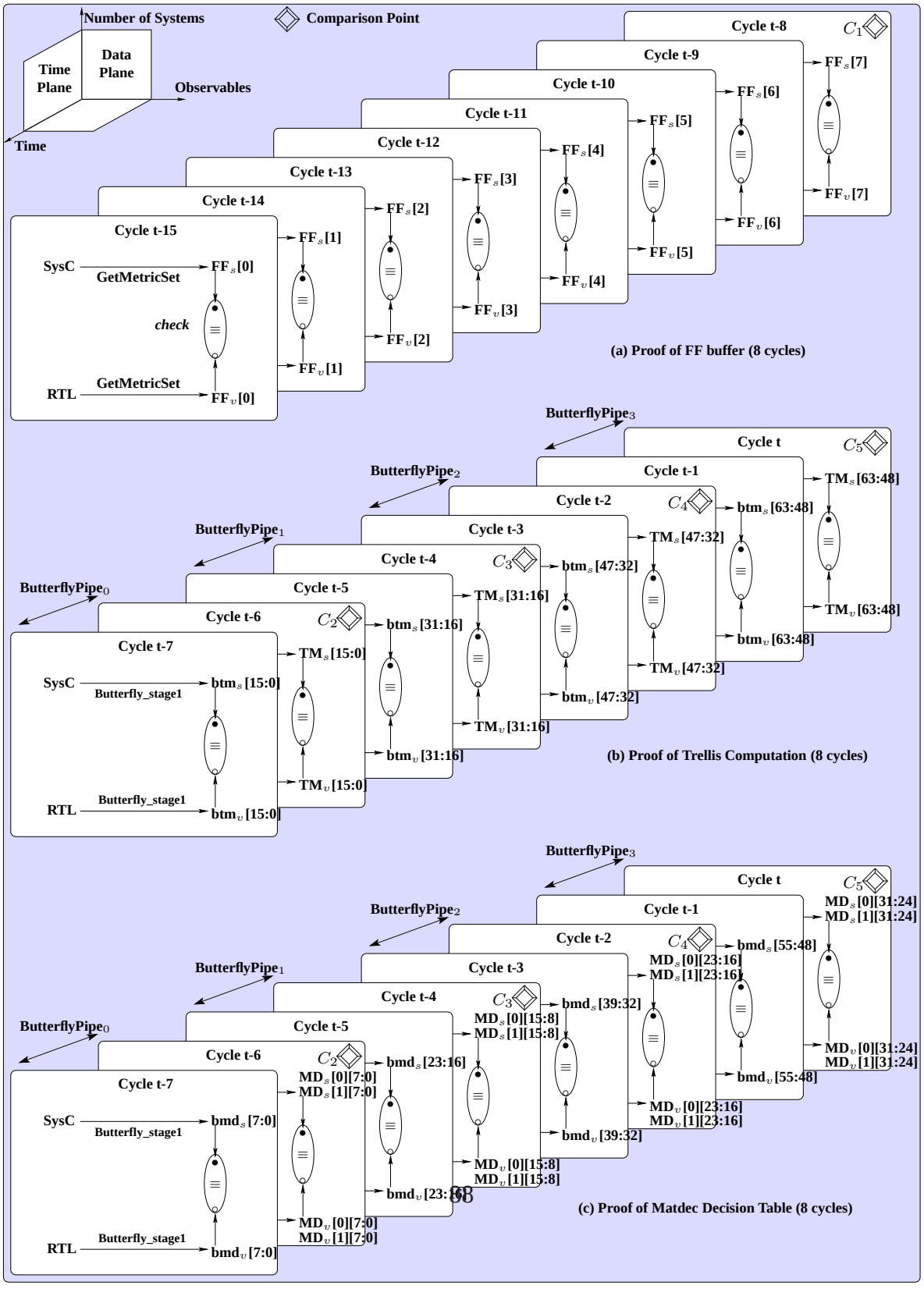


Figure 4.7. Proof of sequential equivalence checking of pipelined Verilog Viterbi design with area optimizations against System C design

Block/Function	Number of clauses in the CNF formula
PLUS	448
LESSTHAN	32
Trellis Condition in the Butterfly	14336
Trellis computation in each stage of butterfly	28672
Trellis per butterfly	57344
MatDec each stage of butterfly	896
MatDec per butterfly	1792

Table A

Design	Number of clauses in the CNF formula
Monolithic Trellis	1892352
RTL decomposition (Design 1)	59136
RTL decomposition (Design 2)	59136

Table B

Block/Function	Number of variables	Number of Symbolic variables generated
PLUS	64	2
Butterfly	128	66
Trellis (monolothic)	2304	2112
Trellis (decomposed)	128	66

Table C

Figure 4.8. Breakdown of number of variables and clauses in the CNF input to zChaff for different blocks.

Chapter 5

Antecedent Conditioned Slicing

5.1 Introduction

We introduce a technique to verify hardware designs by creating design abstractions at the RTL. This abstraction technique involves deletion of statements from the HDL describing the design, and is a derivative of program slicing [245, 247] applied to software programs. The slicing technique removes statements from a program depending on the satisfaction of a given condition, and is thereby called conditioned slicing. Conditioned slicing is defined for HDLs in this work along the same lines as program analysis techniques are defined for software programs, with respect to the Verilog program's control flow graph. [1]he variation of control flow graph for Verilog is called a process dependence graph. Details are provided in the rest of the chapter.

Property specifications written in temporal logics can often be expressed in the form $G(antecedent => consequent)$, which asserts that over all time, under the conditions that the antecedent expression is true, the consequent expression will also be true. The antecedent is a typically a predicate logic formula. In order to check that the property is satisfied, it is sufficient to check only those executions of the system

[1]T

where the antecedent is true. We prune the design with respect to the antecedent in a linear temporal logic (LTL) [163, 171] formula, using the antecedent as the condition in conditoned slicing. We call this technique *antecedent conditioned slicing*.

In order to slice the statements of the RTL design, the truth of the antecedent has to be determined at every program point. This is accomplished by *high level symbolic simulation* of the RTL design. The antecedent formula (expression) is compared with the symbolic expression at each program point. In order to compare the two high level expressions, a rewriting engine is used. If the rewriting engine declares that a sequence of statements (program path) until a particular program point cannot be executed when the antecedent is true, the path is eliminated from the control flow graph of the program. The sliced control flow graph is now much smaller than the original. The sliced design is now verified by passing it through a lower level engine. In this case, the *decision procedure* is a model checker. Although the rewriting engine could have been used as the decision procedure to check the entire LTL formula, the drift of this work is to avail of the sophisticated model checking algorithms that have been researched over a couple of decades. The RTL analysis, can then be viewed vas a "preprocessing" phase by the model checker, that provides it a smaller, more tractable state space than the original. Many abstraction techniques have been tried for bolstering the performance of model checking, but they have all been at the Boolean level for hardware. More details are provided in the next section. Property specific abstractions like slicing that manipulate the program text are more applicable at the source code level in RTL, due to its likeness to software programs. This technique would be a part of the Verilog front end of model checkers, and would create a reduced design at the pre-encoding level, before the model checkers flatten the hierarchy and work on the state transition graph.

A non-trivial example is verified to illustrate the benefits of this technique. An open-source implementation in Verilog RTL of the USB 2.0 core was verified using properties from its specification document.

The rest of this chapter is organized as follows. Section 5.2 gives a background model checking, and provides a short sketch of abstractions in model checking. Section 5.3 provides some details about slicing techniques and conditioned slicing that is necessary for understanding our technique. Section 5.4 show the dependence graph based techniques to obtain a conditioned slice. Section 5.5 extends conditioned slicing to HDLs. In Section 5.6, we introduce and describe antecedent conditioned slicing using examples. Subsection 5.6.2 gives an algorithm for computing antecedent conditioned slices, while Subsection 5.6.3 gives the theoretical basis for our abstractions. In Section 5.7, we provide the experimental results using our technique. We discuss the applications and limitations of our technique in Section 5.8. We summarize with Section 5.9.

5.2 Abstraction Techniques for Model Checking

Model checking is an automatic technique used to formally verify programs [58]. Temporal logic model checking typically requires a formal description of the model whose correctness needs to be established and a property specified in temporal logic. The main temporal logics used to specify properties are LTL [163, 171] CTL and CTL* [58].

In order to make model checking practically feasible, it is necessary to reduce the sizes of these models so that the computations have reasonable time and space requirements. It is also essential that the reduced models retain sufficient information to produce the same results as the original models, with respect to the properties being checked. These two requirements need to be balanced while creating these reduced models, i.e., while generating the *abstract* models from the original or *concrete* models. The process of constructing the abstract model from the concrete one is called *abstraction*. Abstractions are emerging as the key candidates for program verification using model checking.

Abstractions can be performed on the Kripke structure (state-transition model) of a program as well as on the program's source code. Abstraction techniques on Kripke structures are symmetry reduction [89], partial order reduction [56], cone of influence reduction [155], parameterization, compositionality etc. Since the state space of even small programs can be extremely large, it may not be possible to build the Kripke structure for any reasonably sized program. In contrast, abstractions formed by static program analysis will scale well with program size, and are of high economic interest. We focus on these abstraction techniques based on *program transformations* in the rest of the paper.

Abstractions have been used extensively to reduce the computational complexity of model checking. The abstractions are *property preserving* [165].

This implies that given a program and a property to be verified, the In this paper, we will mostly deal with abstractions that are created from t satisfaction of the property in the abstract program implies the satisfaction of the property in the concrete program. Property preservation can be *weak* or *strong* . Weak property preservation can be defined using the branching time μ-calculus defined in [152]. Weak property preservation preserves the truth of properties from the abstraction to its concrete model.

A function α from the powerset of states of a state transition system S_1 to the powerset of states of another system S_2 is said to weakly preserve a property f, if for any state of S_1 that satisfies f, the states in S_2 also satisfy f.

If the converse is also true, then α is said to strongly preserve f. As a result, both truth and falsehood of properties are preserved from the abstraction to its concrete model. Diagnostic counter-examples are "carried over" to the concrete model.

Strong property preservation puts a lower bound on the size of suitable abstractions. Abstractions that result in strong property preservations are typically difficult to construct.

There exist two closely related frameworks for developing abstractions and proving their correctness. Simulation, [178, 193] is about structural relation between abstract and concrete transition systems, representing the step relation of programs by means of an abstraction relation between abstract and concrete sets of states. Each concrete transition must be simulatable by an abstract transition.

Abstract interpretation [70] is the relation between concrete and abstract states by an abstraction function α from concrete sets of states to the smallest element of some abstract property lattice, which represents all the elements of the concrete sets. With α is associated γ, a concretization function, which associates with each abstract element the set of all concrete states represented by it, such that (α, γ) makes a Galois connection.

A difference between these two frameworks is that in the abstract interpretation framework, the computation of the abstract property is given emphasis. The theory of abstract interpretation formalizes the notion of *approximations*. The computation of abstract systems in simulation, however, does not lay emphasis on the precision of abstractions. In fact, simulation was proposed only in the context of strong preservation of properties.

Abstraction techniques are methods that can be used to construct abstractions from the concrete models. The decision as to which details need to be included in the abstraction (for the verification task) can be made manually or automatically.

Manual techniques include user chosen abstract interpretations. The manual construction of safe abstractions trades automation for generality, in the sense that no restrictions are imposed on the class of large (infinite) state systems amenable to the method. The abstractions considered are usually weakly preserving, which means that properties are preserved only from the abstraction to its concrete model. As a result, only the truth of properties is guaranteed to be preserved. The weak requirement guarantees

the existence (at least for universal properties) of finite-state and arbitrarily small weakly preserving abstractions for any concrete model. Users are free to use their 'intuition' for the behavior of the concrete system to come up with an abstraction that they find suitable. There is no *a priori* guarantee regarding the number or the type of properties that can be verified using the abstraction. The appropriateness of the abstractions must be investigated by trial and error. The obligation of proving the safety of the abstraction, in general cannot be automated. Instead, the proof is done manually or with support from theorem provers.

Automatic construction of abstraction systems is more ambitious. Finite-state strongly preserving abstractions (which are small enough for model checking) can only be automatically constructed for restricted kinds of large (infinite) state models. One such well-studied restricted domain is real-time systems, where continuous time gives rise to an infinite state space [14]. Only the systems whose behavior is guarded by certain linear constraint systems on the 'clock' variables are guaranteed to have finite-state strongly preserving abstractions that can be constructed automatically. Almost all existing model checkers for dense reactive systems (real time or hybrid) are based on automatically constructed strongly preserving abstractions. The idea is to let the abstract states be equivalence classes of concrete states, with respect to some behavioral equivalence or equivalence with respect to a property.

An abstraction technique is said to be *sound* if the abstract program is always guaranteed to be a conservative approximation of (*i.e.,* simulates) the original with respect to a set of specification properties. This means that a property holds in the original program if it holds in the abstraction. An algorithm is said to be *exact*, if the abstraction it constructs is bisimilar to the original program. This means that a property holds in the original program if it holds in the abstraction, and the converse is also true. An abstraction technique is *complete* if the algorithm will always find a finite state abstract program for the original program, if one exists. In terms of simulation, if the state-transition graph of the original

program has a finite simulation quotient, then the algorithm can produce a finite simulation equivalent abstract program.

Abstractions can be organized broadly into data, control, configuration or communication abstractions. Data abstractions operate on data values and operations on the data. Control abstractions operate on the order of the operations within a process. Configuration and communication abstractions operate on the order of processes in a program as well on as the communication between programs.

5.2.1 Data Abstractions

Data abstractions abstract away some of the data information for creating smaller models. The abstract model can be derived from some high level description of the program like the program text. Data abstractions are typically manual abstractions. In [64], the abstract transition systems are obtained by computing the abstractions of primitive operators as defined previously.

The programs are modeled as transition systems where states are n-tuples of variable values. The set of all program states is expressed as $D_1 \times D_2 \times \ldots D_n$. A surjective function h maps each D_i onto a set of abstract values. The surjection then maps program states to abstract states. The resulting abstraction (called minimal abstraction by the authors) is approximated by statically analyzing the text of the program. These abstractions show weak property preservation. The approximated abstract model may demonstrate more behaviors than the concrete model, but it is easier to build and verify. The abstractions proposed in this work include arithmetic operation abstractions (such as congruence modulo an integer), single bit abstractions for dealing with bitwise logical operations, product abstractions for combining the effect of abstractions, and symbolic abstractions.

For verifying programs involving arithmetic operations, congruence modulo a specific integer may

be a useful abstraction. Thus, for any $h(i)$, i is replaced by $i \, mod \, m$. The values of an expression are, therefore, constrained to a smaller range, depending on m. When comparing the orders of magnitude of some quantities, the logarithmic representation is used instead of the actual data value. For programs that have bitwise logical operations, a large bit vector is abstracted to a single bit value, according to some function such as a parity generator. Symbolic abstractions can be used in situations where the enumeration of the data values is cumbersome. If the data value is changed to a symbolic parameter, there can be significant savings.

Kurshan [154] uses a similar framework for data abstractions for finite state system verification. The abstractions in this case, however, are not approximated by static analysis of the program text. Instead, they are computed as language homomorphisms in the algebra. The homomorphisms are specified by the user between the actual and the abstract processes, but are checked automatically.

In all the above abstractions, the infinite behavior of the system, resulting from the presence of variables with infinite domains, is abstracted. The abstractions are computed by means of abstract data types. For each variable to be abstracted, the abstract domain and the operations on the sets representable in the abstract domain are defined. An abstract model is then obtained by replacing each variable by one in the abstract domain and each operation by an abstract one. These are general predefined abstractions that are not aimed at specific properties.

Some variations to the above data abstraction techniques are found in [29]. Here, the focus is on properties of single states or transition state-pairs rather than execution sequences. The abstractions use *variable restriction* that eliminates certain variables. The data types of each eliminated variable are abstracted to a single value. This work also constructs abstractions with respect to a single property, as opposed to the generic abstractions constructed by other methods.

Data independent systems are those where the data values do not affect the control flow of the computation. In these cases, the datapath can be abstracted away entirely [201].

In hardware verification, important abstraction techniques use data abstractions [183] and uninterpreted function symbols [38].

5.2.2 Abstract interpretation based abstractions

Many abstraction techniques can be viewed as applications of the abstract interpretation framework [69, 71, 216].

The abstract interpretation framework establishes a methodology based on rigorous semantics for constructing abstractions that overapproximate the behavior of the program, so that every behavior in the program is covered by a corresponding abstract execution. Thus, the abstract behaviors can be exhaustively checked for an invariant in temporal logic. In abstract interpretation, abstractions are usually defined *a priori* for a particular type of analysis. The abstract version of the language semantics is constructed once with manual assistance. Producing a new abstract semantics for on-the-fly verification is a non-trivial task. Some tools such as Cospan [154, 155] and Bandera [67, 87] try to do this task automatically.

Although data abstractions can be included in the realm of abstract interpretations, they form only a part of the possible abstract interpretations. Also, they may not hold over all of the system's execution semantics. Informally, abstract interpretation has a domain of abstract values, an abstraction function mapping concrete program values to abstract values and a set of abstract operations.

Some common abstract interpretations are briefly described for providing a flavor of the technique. *Sign abstraction* consists of replacing integers by their sign and ignoring their actual value. Such ab-

stractions may be useful if there is a proposition like $x = 0$ for some integer x. Since all the information about x is not required, the sign abstraction may be applied, which keeps track of whether x is greater than (`gt`), less than (`lt`) or equal to zero (`eq`). The powerset of the abstract domain is now $\{$`gt, lt, 0`$\}$. Now, all the primitive operations are defined over this abstract domain, such that they satisfy every program execution.

Another common abstraction is *interval abstraction*, that approximates a set of integers by its maximal and minimal values. Thus, if a counter variable appears in a property, the counter can be replaced by the lower and upper limits of the counter.

Relational abstractions retain the relationship between a set of data values. For instance, a set of integers can be approximated by its convex hull. Abstract interpretations of functions is done by maintaining the parameter and the result (signature) of the function in the abstract domain. Fixpoint iteration can also be thought of as an abstraction. A detailed treatment of these abstract interpretations can be found in [68]. Abstract interpretation for μ-calculus is shown in [76, 77].

Since abstract interpretations are not specific to any given property or for any given program, they have the power of generality. The technique considers predefined specifications for all possible programs of a given language. However, this results in the practical problem of computing these abstract interpretations during static analysis, before the verification task begins. The theory of abstract interpretation is mainly concerned with soundness, and not completeness.

5.2.3 Counterexample Guided Refinement

A highly researched field is abstraction refinement techniques for model checking [160, 191, 192]. These techniques are typically automatic abstraction techniques. Counterexample guided abstraction

refinement techniques have been widely studied. An approximation of the set of states that lie on a path from the initial state to a bad state is successively refined. The refinement is done by forward or backward passes, where each pass uses (or refines) the approximation computed by the previous pass. This process is repeated until a fixpoint is reached. If the resulting set of states is empty, the property is proven, since no bad state is reachable. Otherwise, the method does not guarantee that the counterexample trace found is genuine. In other words, the counterexample could be spurious due to the overapproximations. A heuristic is used to find a subset of the reachable states from the initial states. If there is a match, the error is genuine and can be reported as a bug.

Cho *et al.* [112] propose symbolic forward reachability algorithms that induce an overapproximation. The state bits are partitioned into mutually disjoint subsets and do a symbolic forward propagation on individual subsets. Some approaches also use symbolic backward reachability analysis [45]. Govindaraju and Dill [102, 103] allow for overlapping subsets as opposed to the mutually disjoint ones, and present a more refined approximation as compared to earlier schemes.

A model checker that uses upper and lower approximations to verify properties in temporal logic was proposed in [164]. These approximation techniques guarantee completeness without rechecking the model after each refinement. A similar approach has been described in [22], and in Kurshan's localization reduction [155]. These are iterative techniques that are based on the *variable dependency graph*. The localization reduction either leaves a variable unchanged, or replaces it by a non-deterministic assignment.

5.2.3.1 Predicate Abstraction

This technique was first introduced by Graf and Saidi [105]. The predicates are related to the property that is being verified and are automatically extracted from the program text. Das and Dill [81] use this technique with some variation to formally verify complex systems. While Graf and Saidi used monomials to represent the abstract state space, this work uses Binary Decision Diagrams (BDDs) as the representation. Clarke et al describe a related technique in [62]. This work is based on *atomic formulas* that correspond to the predicates, but are used to construct an abstraction function. The abstraction function maintains a relationship between the formulas instead of treating them as individual propositions. The authors also introduce symbolic algorithms to determine if the abstract counterexamples are spurious. If a counterexample is spurious, the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model is identified. The last abstract state in this prefix (the failure state) needs to be split into fewer abstract states by refining the equivalence classes in such a way that the spurious counterexample is eliminated. The extension of these forward algorithms for analyzing counterexamples to backward algorithms that do the same, are found in [27]. This can lead to completely different abstractions. This technique can also handle loop unfolding.

A related approach that performs predicate abstractions by syntactic program transformations automatically is presented in [185]. The algorithm starts from the predicates in the specification formula. Predicates of the original boolean program are represented by Boolean variables in the abstraction. To preserve the correspondence between the predicate and the Boolean variables, the weakest precondition is calculated syntactically, and the Boolean variables are updated iteratively. This method of constructing abstractions is sound and complete. For programs with bounded non-determinacy, the algorithm does not need manual intervention using theorem provers to compute the abstract program, as other predicate abstraction based methods do.

5.2.4 Property specific abstraction techniques

Some abstraction techniques that are highly automatic, but very specific to the program and the property being verified, are *variable hiding* [78] and program slicing [247].

5.2.4.1 Variable Hiding

Variable hiding is a powerful program transformation technique that was used in [78, 122] for verifying C programs. This is an iterative refinement technique that creates overapproximations. In the first iteration, all assignments and function calls that are irrelevant to the property being verified are replaced with a no-op. All conditional choices that refer to irrelevant statements in the program are replaced by *nondeterministic choices*. The use of nondeterminism is a standard reduction technique that can be used to make a model more general. The nondeterminism tells the model checker that instead one specific computation, all possible outcomes of a choice should be considered equally possible. The original computation of the system is preserved as one of the possible abstracted computations, and the scope of the verification is therefore not restricted. If no property violation exists in the reduced system, we can safely conclude that no property violation can exist in the original application.

The resulting abstraction has weak property preservation. It is possible, for instance, that the full expansion of an error trace for a property violation detected in the abstraction does not correspond to a valid execution of the original application. If this happens, it constitutes a proof that the abstraction was too coarse. In that case, the counterexample generated provides clues for including some more statements to make the abstraction less coarse. Typically a few iterations of this type suffice to converge on a stable definition of an abstraction that can be used to extract a verifiable model from a program text.

An example of variable hiding is illustrated. A piece of code and the program transformations that are

generated by variable hiding are given below.

```
h = A[i];
r = r + (++A[i]);
res = r + h;
if (r > MAX)
        {
        m++;
        r = 0;
        }
```

If the property involves `m` or `h`, we obtain the following abstraction after variable hiding. The non-determinism is introduced in the conditional statement, where the variables `r` or `MAX` are not present in the property.

```
h = A[i];
if (NONDET)
        {
        m++;
        }
```

An important abstraction paradigm is *program slicing* [245, 247]. We explore this technique in some detail in the next section, in the context of model checking.

5.3 Slicing Techniques

Program slicing, introduced by Weiser [245, 247] is a widely used abstraction technique used to statically analyze programs and retain parts of the source code relevant to the application [97, 246]. Program slicing (or static slicing) has been used for a variety of tasks, including debugging [246], maintenance [97] and testing [86] within a behavioral domain.

Program slicing has been applied to various software engineering disciplines where manipulation of large programs, and hence their decomposition is desirable. In the past, program slicing has been extended to HDLs [59, 130, 239]. Program slicing has also been successfully applied to hardware verification by Clarke et al [59, 240]. A variation of static program slicing, that improves over it is *conditioned slicing* [46, 48]. Conditioned slicing augments static program slicing by specifying some initial states of interest in the slicing criterion. Conditioned slicing has been shown to produce smaller and more meaningful abstractions than static slicing. Conditioned slicing has been used for program comprehension [167], reuse [48], migration [47] and re-engineering [49].

5.3.1 Static Slicing

Slicing, in the most general sense, is a program transformation involving statement deletion, that preserves some projection of the semantics of the original program. The aspect of the program that must be preserved is application specific, and is captured by the slicing criterion. We present here, some necessary background for program slicing.

Definition 5.3.1. Static slicing criterion

A slicing criterion of a program P with an input alphabet Σ, is a pair $\langle i, V \rangle$ such that i is a program point in P and $V \subseteq \Sigma$.

A set of statements I_s is said to *affect* the values of V at i in a given slicing criterion $\langle i, V \rangle$, if I_s defines a subset of V that is used in i.

Definition 5.3.2. Static slice for programs

A slice S of a program P on a slicing criterion $\langle i, V \rangle$ is a subset of the statements of P that might affect the values of V at i.

A static slice preserves the projection of the semantics of the original program for every possible execution of the program. This can result in very large slices [150, 167]. Many slicing algorithms have been proposed as variations of static slicing, to create smaller slices [150, 230, 243]. A detailed survey of program slicing techniques can be found in [231].

5.3.2 Conditioned slicing

Canfora et al [46] introduced the notion of *conditioned slicing*, that forms a theoretical bridge between static and dynamic slicing. Conditioned slicing augments static slicing by introducing a condition that specifies the initial set of states in the criterion. It does not give specific inputs, unlike dynamic slicing. This slicing technique, therefore allows slicing with respect to the initial constraints in the program. We present some basic definitions of conditioned slicing that appear in the literature.

Definition 5.3.3. Conditioned slicing criterion

Let Σ be the input alphabet of the program P. Let C be a first order predicate logical formula on the variables in Σ. A conditioned slicing criterion is a triple $\langle C, i, V \rangle$, where i is a statement in the program, and $V \subseteq \Sigma$.

Definition 5.3.4. Conditioned slicing for programs

A conditioned slice of a program P on a conditioned slicing criterion $\langle C, i, V \rangle$ consists of all the statements and predicates of P that might affect the values of the variables in V at i, when the condition C holds true.

Tip [230] introduced a more restricted form of conditioned slicing called constraint based slicing. In all these cases, the *condition* that specifies the set of initial states, and is used for slicing is a first order predicate logic formula.

In situations where the initial set of constraints for the program analysis are known, this technique can be employed to get much smaller slices than those produced by static slicing.

Conditioned slicing has been automated with significant success on C and WSL code [79, 80]. More recently, analogous to backward and forward slicing [205], backward and forward conditioning were introduced [95]. In forward conditioning, a statement is deleted if, when the condition is satisfied, it cannot be executed. In backward conditioning, a statement is deleted if, when executed, it cannot lead to the condition being satisfied. In this paper, wherever it appears, conditioning refers to forward conditioning.

5.4 Dependence graph based slicing

Ottenstein and Ottenstein define slicing as a reachability problem in a dependence graph representation of the program [189]. They use Program Dependence Graphs (PDGs) [93] for static slicing of single procedure programs. The statements and predicates of a program correspond to the vertices of the PDG and the edges correspond to data and control dependences between statements. In dependence graph based slicing approaches, the slicing criterion is identified with a vertex v in the PDG. The i in the slicing criterion $\langle i, V \rangle$ corresponds to v in the PDG, while V stands for the set of all variables defined or used at v.

For slicing of multi-procedure programs, System Dependence Graphs (SDGS) were introduced [124]. An SDG combines the *procedure dependence graphs* of all the called procedures of a program, along with the *program dependence graph* of the main program by allowing edges that can model procedure calls.

Let the program dependence graph for a program P, denoted by G_P be a directed graph.

Definition 5.4.1. Control Dependence Edge

A control dependence edge from v_1 to v_2, where v_1 is a predicate vertex, denoted by $v_1 \longrightarrow_c v_2$, implies that the truth value of the predicate expression represented by v_1 determines whether or not v_2 is executed.

Definition 5.4.2. Flow Dependence Edge

A flow dependence edge from v_1 to v_2, denoted by $v_1 \longrightarrow_f v_2$, implies that there is some variable x, that is defined at v_1 and used at v_2 and there is an execution path from v_1 to v_2 along which, there is no assignment to x.

5.4.1 Conditioned slicing using PDGs

We present here, conditioned slicing, using a dependence graph approach. To the best of our knowledge, such a treatment of conditioned slicing has not been shown before.

Figure 5.1 shows an example program written in pseudo-code. The PDG for the program is shown in Figure 5.2. In order to find a static slice for the program with respect to slicing criterion $\langle 11, B\rangle$, we find all the reaching definitions of B at node 11. Then, we find the set of all reachable nodes from these nodes in the PDG. This set $\{1, 2, 3, 4, 6, 7, 9\}$, gives us the desired static slice. The nodes are shown in bold in the figure. set of ALL reachable may be wrong write vertices instead of nodes

Now, consider the conditioned slicing of the program with respect to the slicing criterion $\langle C, 11, B\rangle$, where C corresponds to the predicate $(N < 0)$. To obtain the conditioned slice for a given predicate C, we *project* the PDG with respect to the predicate, and then use the static slicing algorithm on the projected PDG. Figure 5.3 shows the application of this technique to obtain the conditioned slice for the mentioned criterion. Initially, all the vertices in the graph are drawn dotted. All the statements that would get executed when the predicate C is satisfied, are marked, and the corresponding vertices in the graph are made solid. The graph is then traversed only for solid (marked) vertices. The set of all vertices reached during the traversal are made bold. This set $\{1, 2, 3, 4\}$ gives the desired conditioned slice. The conditioned slice, therefore, contains only those statements that get executed when C evaluates to *true*. It is evident from this example that the conditioned slice is typically much smaller than the static slice.

5.5 Conditioned slicing for HDLs

We present here, an extension of conditioned slicing to HDLs. The HDL computational paradigm differs fundamentally from traditional languages, since HDLs model non-halting, reactive systems with concurrently running processes. The processes are not explicitly called, as they are in a program's procedures. Instead, they communicate through signal dependency [59]. In order to extend conditioned slicing for HDLs, we use a few definitions from earlier work in program slicing [247] and its extension to HDLs [59, 239] with minor modifications. We also extend the definitions from earlier work in conditioned slicing [46, 48].

Let M be a Verilog program [2] with k concurrent processes P_i, such that $M = \|_{i=1}^{k} P_i$, where $\|$ is the parallel composition operator [239]. The processes communicate with each other through shared signals. The sensitivity list of a process p is the list of shared signals that can cause the "calling" or re-evaluation of p. When a signal in the sensitivity list gets assigned, p is re-evaluated or called with the new value of the signal.

Definition 5.5.1. Signal dependence

A process region P is said to be signal dependent on a statement i, if i assigns a value to a signal which is present in the sensitivity list of P. erilog

Definition 5.5.2. Inter-process CFG

[2] The term "program" is used with the same meaning as in [59] for VHDL designs. A Verilog program is a set of concurrent processes that executes as a series of simulation cycles.

An inter-process control flow graph G for a Verilog program is the structure $\langle G_1, G_2, \ldots G_k, Esd \rangle$ where $G_1, G_2 \ldots G_k$ are the control flow graphs representing the processes in the program, and E_{sd} is the set of edges representing the signal dependencies between the processes.

Definition 5.5.3. Inter-process SDG

An inter-process def/use graph for a V program is a structure $\langle G, \Sigma, D, U \rangle$, where Σ is the set of signals in the program, D is a function mapping the nodes of G to $\Delta(\Sigma)$ and U is a function mapping the nodes of G to $\gamma(\Sigma)$, where $\Delta(\Sigma)$ is the set of signals defined and $\gamma(\Sigma)$ is the set of signals used in the statements corresponding to the nodes.

This graph is also called a system dependence graph. Program slicing of HDLs has been modeled as a dependence graph node reachability problem in [59]. Each concurrent process has a corresponding PDG. The PDG for each process is modified for HDLs by making provision for inter-process communication through signals. Implicit procedure calls are generated in the inter-process control flow graph, whenever a signal is potentially assigned. The PDGs of the processes are connected appropriately to form a System Dependence Graph (SDG). The slices are computed by following the chains of dependences represented in the edges of the SDG. We extend the node reachability problem to include conditioning of HDLs.

Definition 5.5.4. Static slicing for HDLs

An inter-process slice S_p within a Verilog program M, on a given criterion $\langle i, V \rangle$ is an executable subset of M obtained recursively containing all the statements that may affect the values of V at i within the process P in which i is defined, and all the slices on the slice criterion $\langle i_s, V_s \rangle$ where i_s is the set of statements defining the set of signal V_s on which process P is dependent.

Definition 5.5.5. Conditioned process

A conditioned process $P(C)$, for a process P and a condition C, is an entity containing the set of statements of P that can be executed when C holds true.

v

Definition 5.5.6. Conditioned program

A conditioned program with respect to the condition C is represented as $M(C) \equiv \|_{i=1}^{k} P(C)_i$ where $P(C)_i$ is a conditioned process.

Definition 5.5.7. Inter-process conditioned slicing

An inter-process conditioned slice S_c for a Verilog program M, on a given criterion $\langle C, i, V \rangle$ is a subset of M obtained recursively containing
a) all statements in the scope of condition C, that affect the values of V at i within the process P where i is defined
b) all the conditioned slices with respect to the slice criterion $\langle C, i_s, V_s \rangle$, where i_s is the set of statements that define the set of signals V_s on which P is signal dependent.

The conditioned slice of a Verilog program M can be computed using its SDG representation. The condition C in the slicing criterion is applied to all the PDGs of all the processes P_i in M. The resulting conditioned process $P(C)_i$ is marked in its PDG. The conditioned program $M(C)$, is thus obtained from all the conditioned processes. $M(C)$ is now marked on the SDG. The conditioned slices are computed by finding the transitive closure of the control, flow and signal dependencies of the conditioned program, $M(C)$ in the SDG.

Figure 5.4 illustrates an example program of a hardware system, in Verilog HDL. The three processes `P1,P2` and `P3` correspond to the concurrently executing `always` blocks in Verilog. The parentheses of the always blocks list the signals on which each process is dependent. Now, consider the slicing criterion $\langle C, end_{P1}, result \rangle$, where C corresponds to the predicate $valid = true$ and end_{P1} corresponds to the end statement of process $P1$. We apply this predicate on each of the three processes, to obtain the corresponding conditioned processes, $P(C)_1$, $P(C)_2$ and $P(C)_3$. The resulting conditioned program, $M(C)$, is shown in Figure 5.5. The portions of the code which can be executed, when the predicate $valid$ is true, are shown. The conditioned program is now analyzed for determining the slice with respect to the given criterion. The transitive closure of the data, control and signal dependencies of the relevant variables yields the program of Figure 5.6. It can be seen that `P2` is included in the slice due to the signal dependence of `P1` on `P2` with respect to the variable $reset$. The statement in `P3` is eliminated, since the variables in the slicing criterion do not have any dependencies on it. However, the process stub is maintained, in order that the behavior of the slice with respect to fairness constraints remains the same as the original program.

5.6 Antecedent conditioned slicing

We use conditioned slicing for verification of hardware designs described in Verilog HDL. Our technique aims at reducing state space of the design, by slicing away the parts of the design irrelevant to the property being verified. We assume that the properties are specified as temporal logic formulae specified in LTL.

For these properties, we can use the antecedent to specify the set of initial states that we are interested in. *The antecedent therefore, forms the condition in the slicing criterion.* All the statements that would

get executed when the antecedent is true (or the condition is satisfied) are included in the slice. The statements on the paths that cannot get executed when the antecedent is false, are removed. The reduced program still preserves its behavior *with respect to the property being checked*. We therefore create property preserving abstractions using conditioned slicing.

All prior art in verification using program slicing uses static program slicing techniques. While slicing property specifications written in temporal logic, these techniques retain the set of all statements of the program where the antecedent is true, *as well as those where it is not*. This is because static slicing retains all possible executions of the relevant variables.

However, in property based verification, *we do not need to check the states where the antecedent is false.* In these cases, static slices might be too large and include statements that are not of interest. We introduce a precise abstraction on the basis of conditioned slicing, *antecedent conditioned slices*.

The key idea in these abstractions is that they utilize information from the antecedent. Antecedent conditioned slicing, therefore, forms more meaningful abstractions than static slicing and also produces smaller slices that can reduce the verification state space significantly. We describe them in detail in the next section.

5.6.1 Computing antecedent conditioned slices

In our property language, we permit LTL properties h of the form

$$G(a \Rightarrow c)$$
$$G(a \Rightarrow X^{=n}c)$$

where a and c are propositional formulas, and $X^{=n}q$ means at distance $n \geq 0$, q holds, *i.e* X applied n times to q.

We also permit bounded LTL properties, that we represent by $[h]^k$ for a given bound $k \geq 1$. We detail the theoretical basis for bounded LTL properties, since we want to justify our experiments that were performed using a bounded model checker. We permit bounded properties of the form

$$[G(a \Rightarrow c)]^k$$
$$[G(a \Rightarrow X^{=n}c]^k$$
$$[G(a \Rightarrow Fc]^k$$

5.6.2 Algorithm for antecedent conditioned slicing

In [225], we presented the algorithm for antecedent conditioned slicing at a conceptual level. In Figure 5.7, we present a more detailed version of the algorithm that provides a platform for the instruction slicing algorithm for processor verification.

The procedure $antecedent\_conditioned\_slice()$ takes a Verilog program and an LTL property h as input. If the property is of the type $G(a \Rightarrow X^{=n}c)$, where $n = 0$ and a and c are propositional formulas, the $get\_conditioned\_slice()$ procedure is called. The $get\_conditioned\_slice()$ procedure returns a *marked* Verilog program. The marked program is now pruned. The details of the marking and pruning procedures are explained further in the section. If the property is of the form $a \Rightarrow X^n c$, where $n > 0$, the program is "unrolled" n steps into the future. The $get\_conditioned\_slice()$ procedure is called repeatedly, for every successive "unroll" of the program into the future. Each unroll is annotated with the corresponding value of the time step t. The union (denoted by the symbol $\coprod$) of marked programs returned by every call to $get\_conditioned\_slice()$ gives the desired marked program. (The $\coprod$ of two

marked programs is defined at every statement as $T \coprod T = T$, $F \coprod F = F$, $T \coprod F = T$, $T \coprod X = T$, $F \coprod X = X$, $X \coprod X = X$). The final marked program is pruned. The resulting antecedent conditioned slice is then statically sliced with respect to the slicing criterion $\langle i, V_h \rangle$, where i is the program point where the property is declared and V_h are the variables in the property h. This ensures that all the statements that affect the variables V_h in the antecedent conditioned slice are retained and the others are deleted. The procedure $get\_static\_slice()$ obtains a static program slice as a final step in the antecedent conditioned slicing procedure. Details of the static slicing algorithm for HDLs can be found in [59, 239].

The procedure $get\_conditioned\_slice()$ takes as input, a Verilog program, the time step t, and the slicing criterion, $\langle a, i, V_h \rangle$, where a is the antecedent, i is the program point where the property is declared. and V_h are the variables in the property h. It returns a marked program as an output. The procedure $compute\_fixpoint\_expression()$ returns the symbolic expression for the antecedent in the "current" (k^{th}) cycle after unrolling the program. The $get\_conditioned\_slice()$ procedure at any time t marks the program graph with the truth values of the antecedent at time t. In the case where the property is of the type $a \Rightarrow X^{=n}c$, where $n = 0$, the procedure is called with $t = 0$. The marking procedure is called over all the processes in the program. The function $symbolic\_expression()$ computes the symbolic expression for every statement executed t time cycles after the cycle in which the antecedent is true (k). The $decision\_procedure$ function is then called for the symbolic expression of every statement. This function compares the symbolic expression of a given statement to the symbolic expression for the antecedent using a decision procedure like a rewriting engine or a SAT solver. The procedure can return T, F or X. A statement is marked T in the slice if it is executed t time steps later from the time step k when the antecedent a holds true. A statement is marked F if it *cannot be executed* t time steps later from the time step k when a is true. If the statement does not, in any future, depend on the truth value of a, it is marked X in the slice. In the case where the decision procedure, typically, a rewriting engine, is

not able to determine the truth of the antecedent at a statement, it would also return an X.

The procedure $compute\_fixpoint\_expression()$ takes a Verilog program and the antecedent as input and gives the symbolic expression E_k corresponding to the antecedent as an output. It performs an "unrolling" to eliminate the loops in the *always* blocks of a Verilog program. The antecedent is assumed to be true in the current time cycle k. In the first backward unroll, the program statements that, if executed in the previous ($(k-1)^{th}$) cycle, would cause the antecedent to be true in the current cycle, are isolated. Similarly, in successive unrolls, all the statements in previous cycles that would cause the antecedent to be true in the current cycle are isolated. For each of these statements, the symbolic expression is computed using the $symbolic\_expression()$ function. The symbolic expression of a statement s at time k is a Boolean expression that contains the guards as well as the assignments that are required to execute the statement at time k.

The expression E_k contains the symbolic value of the antecedent in the first step. At the end of a single unroll (iteration in the algorithm), the symbolic expressions of all the statements that define the variables in E_k are computed. These are composed in E_{k-1} to obtain the new expression for the next iteration. When the symbolic expressions start repeating over iterations, a fixpoint is reached. This means that all possible execution paths that can cause the antecedent to be true are included in the final expression, E_k.

The $prune()$ procedure takes a marked program as input and returns a slice S as an output. It inspects every path in the program control flow graph. For any segment on a path, if there is an F on every statement (node in the control flow graph) in the path, until the endpoint or the leaf node of a process, the path is sliced. Otherwise, the path is retained.

In the verification step, the antecedent conditioned slice S_a is passed through a model checker along with the property h. In case the property is not verified, a counterexample is returned.

5.6.3 Verification using antecedent conditioned slicing

We now provide the theoretical basis for verification of antecedent conditioned slices.

Let $M = (S, R, I)$ be the Kripke structure representing a Verilog program P, where S and R represent the states and the transitions and I represents the set of initial states of the program.

Each LTL property h is interpreted over full paths of the underlying structure. Each bounded LTL property $[h]^k$ is interpreted over finite paths of length k in the underlying structure. Then, for M, we define $M \models h$ to mean $\forall$ full paths $x \in M$ starting at any $s_0 \in I$, $M, x \models h$.

For bounded formulas $[h]^k$, we define $M \models [h]^k$ to mean $\forall$ finite paths $x = s_0, s_1 \ldots s_k \in M$ starting at $s_0 \in I$, $M, x \models h$ in the standard temporal semantics for finite timelines.

Let h be a bounded LTL formula of the form $[G(a \Rightarrow c)]^k$. The antecedent conditioned slice of P with respect to the criterion $\langle a, i, V \rangle$ is denoted as $P_{|a}$. From the algorithm, $P_{|a}$ comprises only the set of those states of P, where the antecedent a is true. Let $N = (S', R', I')$ be the Kripke structure representing the Antecedent conditioned slice $P_{|a}$, such that S' and R' represent the states and the transitions and $I' = I$. In general, the slice structure N for M and bound k is the substructure of M comprising $M_{|a}$, or the set of states in M that satisfy a, and those states of M at a distance at most k from $M_{|a}$.

For $h = G(a \Rightarrow c)$, a bound of 0 suffices, and the slice N for M is just $M_{|a}$.

For $h = G(a \Rightarrow X^{=k}c)$, $[G(a \Rightarrow c)]^k$, $G(a \Rightarrow F^{\leq k}c)$, $[G(a \Rightarrow Fc)]^k$, $[G(a \Rightarrow a\, U_s c)]^k$, a bound of k suffices and slice N corresponds to $M_{|a}$ and all the states of M at a distance of at most k from $M_{|a}$.

For the special case when $h = [G(a \Rightarrow X^{=n})]^k$ and $n \leq k$, we can define slice N to consist of $M_{|a}$ and all the states of M at a distance of at most n from $M_{|a}$. We will use this streamlining in our experimental results where n is typically of the order of 2 or 3 and $k = 50$.

In order to prove the correctness of antecedent conditioned slicing, we need to prove that the property h holds on the original program if and only if holds on the antecedent conditioned slice.

We state the correctness theorem for all the LTL formulas discussed above and outline an illustrative proof.

Theorem 5.6.1. $M \models h \iff N \models h$ *where*

$h = [G(a \Rightarrow c)]^k$.

Proof:

We say a state t of M is "close" if there a path x in M from some initial state $s_0 \in I$ to t of length at most k. Let $M \models h$. Then, $N \models h$ due to the following reasoning. All "close" states in M satisfy $a \Rightarrow c$. Since N is a substructure of M, this is also true of all the close states in N.

Now,let $N \models h$. Then, $M \models h$ due to the following reasoning. All close states in N satisfy $a \Rightarrow c$. These states include all the close states of $M_{|a}$. Thus all close states of M that satisfy a must also satisfy $a \Rightarrow c$. All states of M that satisfy $\neg a$, including the close states, satisfy $a \Rightarrow c$ vacuously.

Therefore, we infer that all close states of M satisfy $a \Rightarrow c$. QED.

Theorem 5.6.2. $M \models h \iff N \models h$ *where* h *is one of the following*

$G(a \Rightarrow X^{=k} c)$

$G(a \Rightarrow F^{\leq k} c)$

$[G(a \Rightarrow Fc)]^k$

$[G(a \Rightarrow X^{=n} c)]^k$, $0 \leq n \leq k$

We outline the proof for the correctness of the antecedent conditioned slice for an illustrative formula, where $h = [G(a \Rightarrow Fc)]^k$.

Proof:

We need to prove that $M \models h \iff N \models h$. Let $k\text{-}shell$ be the states reachable from forward or backwards paths of length k from every state in $M_{|a}$. $N = M_{|a} + k\text{-}shell$.

Let $M \models h$. Then, $N \models h$ due to the following reasoning. All "close" states in M satisfy $a \Rightarrow c$. Since N is a substructure of M, this is also true of all the close states in N.

Let $N \models h$. Then, $M \models h$ according to the reasoning that follows. Pick an arbitrary path x of length k in M.

Case 1: $x \in M_{|a}$

All close states in N satisfy $a \Rightarrow Fc$. These states include all the close states of $M_{|a}$. Thus all close states of M that satisfy a must also satisfy $a \Rightarrow Fc$. Therefore all states of x satisfy $a \Rightarrow Fc$.

Case 2: $x \notin M_{|a}$

This means x starts in a state that satisfies $\neg a$. Let, in this case, x remain in $\neg a$ within the $k\text{-}shell$. In this case, the distance is too much to cover in k steps, and x will never reach $M_{|a}$. Therefore, all states of x satisfy $\neg a$, and thereby satisfy $a \Rightarrow Fc$ vacuously.

Case 3: $x \notin M_{|a}$

This means x starts in a state that satisfies $\neg a$. Let, in this case, x not remain in $\neg a$ within the $k\text{-}shell$. Let x exit the $k\text{-}shell$ and enter $M_{|a}$. In this case, by the reasoning in *Case 1*, all states in x satisfy $a \Rightarrow Fc$. The proofs for the other formulas are similar. QED.

5.7 Experimental Results

We provide experimental results on the Verilog RTL implementation of the USB 2.0 Function Core. The USB is a standard interconnect between computers and peripherals. This core operates at full and high speed rates (12 and 480 Mb/s). The source code can be found at [65]. The properties chosen for verification were from the USB 2.0 core specification document [85]. These properties involved the many state machines in the implementation, and were essentially control based properties. The safety and bounded liveness properties expressed in temporal logic have been listed below as LTL formulae , where k is the bound (in this case k = 50) and also explained in English, for the sake of readability. The variables used in the LTL formulae are the signal names in the Verilog code. The Verilog state machines that correspond to the given property are given in parentheses.

- **P1**: $G((state == SPEED\_NEG\_FS) \Rightarrow$
  $X((mode\_hs) \wedge (T1\_gt\_3\_0ms) \Rightarrow$
  $(next\_state == RES\_SUSP)))$

  If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3ms, it will go to the reset/suspend state. (Main State Machine)

- **P2**: $G((state = IDLE) \wedge (ep\_stall) \wedge (pid\_PING) \wedge (mode\_hs) \Rightarrow \neg(token = ACK))$.

  If the machine is in the IDLE state and high speed mode, if a stall is forced, then a PING token is ignored (or an acknowledgment is not sent out.) (Main Protocol State Machine)

- **P3**: $G((tokenout) \wedge (buf0\_na) \wedge (buf1\_na) \Rightarrow$
  $(signal = NACK))$.

If the OUT token is received and both buffers are not available, the NACK handshake is issued. (Main Protocol State Machine)

- **P4**: $G(\neg(suspend\_clr) \Rightarrow$
  $\neg(state = RESUME) \wedge$
  $\neg(state = RESUME\_REQUEST))$.

  If the *suspend* bit is not cleared, then the machine is not resuming from the SUSPEND state. (Main State Machine)

- **P5**: $G((crc5err) \vee \neg(match) \Rightarrow$
  $(state = IDLE) \wedge \neg(send\_token))$.

  If an packet with bad CRC5 is received, or if there is an endpoint field mismatch in the IDLE state, then the token is ignored. (Main Protocol State Machine)

- **P6**: $G((state == CRC1) \wedge (tx\_ready) \Rightarrow$
  $X(tx\_ready \Rightarrow X(state == IDLE)))$

  If the machine is ready to transmit in the CRC state, the IDLE state should be reached after two states. (Transmit/Encode State machine.)

- **P7**: $G((state == OUT2B) \wedge \neg(abort)$
  $\wedge\neg(pid\_sequence\_err) \wedge \neg(no\_bufs) \wedge\neg(to\_small) \wedge \neg(to\_large) \Rightarrow$
  $(token\_pid\_sel\_d == ACK))$

  If the machine is in the OUT state, and it is not aborted, and there is no error in the process id, and buffers are available and the data is not too small or too large, then an acknowledgment token is sent out. (Main Protocol State Machine)

- **P8**: $G((state == SPEED\_NEG\_J) \wedge$
  $(chirp\_cnt\_inc) \wedge (chirp\_cnt == 3'h1) \Rightarrow$
  $F(state == SPEED\_NEG\_HS))$

  If the machine is in the "J" speed negotiation mode and a counter is initialized, then eventually, high speed mode is reached. (Main State Machine)

- **P9**: $G((state == RESUME\_WAIT) \wedge$
  $\neg(idle\_cnt\_clr) \Rightarrow$
  $F(state == NORMAL))$

  If the machine is waiting to resume operation, and a counter is set, eventually (after 100 mS) it will return to normal operation. (Main State Machine)

- **P10**: $G((state == OUT) \wedge (abort) \Rightarrow$
  $X(state == IDLE))$

  In any OUT state, if the *abort* signal is asserted, the machine gets into IDLE mode. (Main Protocol State machine)

- **P11**: $G(((state == SPEED\_NEG\_K) \vee$
  $(state == SPEED\_NEG\_J)) \wedge (se0\_long) \Rightarrow$
  $(XX((T1\_gt\_3\_mS) \wedge (mode\_hs)) \Rightarrow$
  $(state == RES\_SUSP)))$

  If the machine is in the speed negotiation state, and $se0$ is asserted for a long time, then if after two cycles, the machine is in high speed mode and 3 mS have elapsed, the machine will either go into RESET or SUSPEND states. (Main State Machine)

- **P12**: $G(\neg(wb\_req) \Rightarrow$
  $(state == IDLE))$

  If a write-back request is not received, the machine remains in the IDLE state. (Interface State Machine.)

- **P13**: $G((state == IDLE) \wedge (match\_r) \wedge$
  $\neg(ep\_disabled) \wedge \neg(pid\_SOF) \Rightarrow$
  $(state == IDLE))$

  If there is an endpoint mismatch and an SOF token is not received, then the machine remains in the IDLE state. (Main Protocol State Machine)

We used the SMV [177] model checker for our experiments. All experiments were performed using a 3GHz Intel Pentium 4 processor with 1 GB ram. Since the USB is a large design with approximately 10^{331} state elements, model checking this design using SMV alone resulted in memory overflow. Hence, in order to provide a baseline for our technique, we use Bounded Model Checking with a uniform bound of 50 for all properties. The results were generated using the SAT-based BMC utility of the Cadence-SMV tool.

The results of our experiments are presented in Table 5.7. The first and second columns provide the execution times of BMC when running on the original program, and when running on the static slice with respect to a given property. In the third column, we show the performance increase due to the application of conditioned slicing. We can observe that the performance increase in BMC due to static slicing is not very high as compared to the original program. In contrast, there is a tremendous gain in bounded model checking performance due to antecedent conditioned slicing of the design, when

Property Checked	BMC Original	BMC Static Sliced	BMC Conditioned Slicing	Proof Result
P1	11.47	11.02	0.96	Unsat
P2	43.59	41.31	30.37	Unsat
P3	136.3	87.95	44.14	Unsat
P4	27.36	27.02	0.75	Unsat
P5	227.27	201.07	39.27	Unsat
P6	304.51	100.14	0.76	Unsat
P7	68.95	42.69	9.86	Unsat
P8	15.57	8.26	1.17	Sat
P9	19.56	4.8	1.32	Sat
P10	477.53	361.37	13.87	Unsat
P11	2.16	2.01	0.96	Unsat
P12	85.24	81.12	0.7	Unsat
P13	85.49	44.55	4.69	Unsat

Table 5.1. Comparison of execution times (in seconds) taken for verification of properties by the original program, the static slice and the antecedent conditioned slice. A bound of 50 was given to BMC for all experiments.

compared to the original as well as the statically sliced design. We will discuss the import of these results in the next section.

Properties $P1$ to $P7$ and $P10$ to $P13$ are safety properties. $P8$ and $P9$ are liveness properties. Since no counterexamples to these properties are generated within the given bound, the safety properties are partially verified. The liveness properties do not find a sequence that satisfies the property for the given bounds. We increased the bounds on these properties with significant performance gains.

Figure 5.8 shows the performance of some sample properties after applying our algorithm for increasing bounds of BMC. We find that there is a spectacular increase in performance due to antecedent conditioned slicing of the RT-Level design. We placed a reasonable bound of 50 on the number of clock cycles we verified. Communication protocol designs, however, need to be verified for much higher bounds. As the bound increases, the performance gain scales too.

Our algorithm performs well even on safety properties that do not hold and produce a counterexample as well as liveness properties that hold within the given bound.. We do not reproduce them here, since they require different bounds each time.

5.8 Discussions

An important issue we would like to address is the improvement of conditioned slicing over static slicing. Static slicing has been applied to HDL verification before, with performance speedups. However, theoretically, static program slicing has not been shown to be different from the *cone of influence reduction* (COI) used by existing model checkers [155]. Clarke et al [59], while comparing their static slicing technique to COI reductions, mention that COI reduction is similar to building a dependence graph for

the program, and then deciphering relevant variables using graph reachability. The dependence graph may be constructed on the HDL source code (slicing), or on the synthesized netlist. This shows that the only difference between static slicing and COI reductions, is their application domain (pre or post encoding). Semantically, the two are not different. Any performance gains due to static slicing, therefore, are due to the ease of *model generation*, as opposed to that of *model verification*.

In contrast, conditioned slicing creates a different program (or design) from static slicing or COI reductions. The antecedent conditioned slice forms a new entity that does not bear similarity in structure or meaning to the original program, but retains the behavior with respect to the property in question. The performance gains due to antecedent conditioned slicing, therefore, are due to the powerful abstractions created by this technique. Although it can be argued that when this technique is combined with static slicing, the overall performance gain achieved may also include the smaller model generation factor, the most significant gains in performance are primarily due to the reduction in the complexity of model verification.

Our algorithm for antecedent conditioned slicing is entirely structural and based on syntactic transformations. Consequently, it suffers from some inherent weaknesses. For instance, the efficiency of the algorithm depends on the type of property being verified and the structure of the program being verified. In contrast, the effectiveness of an abstraction based on semantic transformations is not property or program dependent. Also, when a property requires reasoning over many time steps, the size of the antecedent conditioned slice increases. In the case of properties where the antecedent changes, all future behavior of the program would need to be retained for each time step. In these cases, the antecedent conditioned slice would not be too much smaller than the original program.

However, when applied to specific practical applications and properties, the abstraction can be very useful in reducing state space. In that respect, processor verification is an excellent application for our

abstractions. The antecedent, which is the instruction word in the single instruction machine, does not change through the duration of the property.

5.9 Summary

In a preliminary version of this work [227],we have shown that our technique is effective for the simple case of safety properties of the form $G(antecedent \Rightarrow consequent)$, where the $antecedent$ and $consequent$ are propositional logic formulas. In this work, we have demonstrated that our abstraction based property verification technique using antecedent conditioned slicing is effective for safety and bounded liveness properties written in temporal logic. Our proposed methodology has been shown to maintain correctness. It lends itself easily to automation, especially to be built on existing model checkers. The accompanying disadvantage is the loss of generality due to the property specific nature of the abstraction. The experimental results show that the technique scales very well, and produces exponential performance speedups when compared to state-of-the-art techniques. The technique therefore seems very promising and can be applied to different domains of hardware and software verification, like verification of microprocessors and device drivers. Future work would focus on these varied verification application domains, where state-of-the-art model checkers are not very efficient.

```
        begin
1:              read(N);
2:              A = 1;


3:              if (N < 0)
                    {
4:                  B = f(A);
5:                  C = g(A);
                    }
                else
6:                   if (N > 0)
                         {
7:                       B = f’(A);
8:                       C = g’(A);
                         }
                     else
                         {
9:                       B = f’’(A);
10:                      C = g’’(A);
                         }
11:             print(B);
12:             print(C);
         end
```


Figure 5.1. Example Program written in pseudo-code

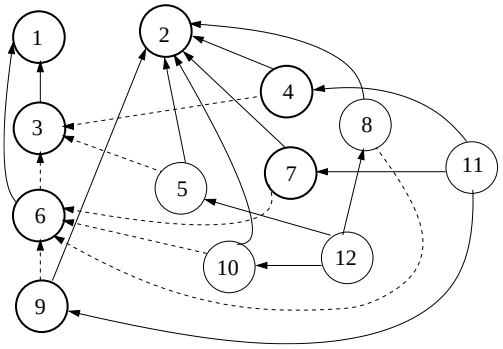


Figure 5.2. Program Dependence Graph of the program. The solid edges denote data dependency and the dashed edges denote control dependencies. The vertices in bold denote the static slice of the program in Figure 5.1 with respect to the variable B at statement 11.

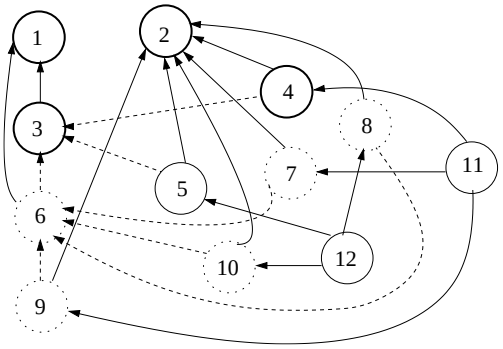


Figure 5.3. Conditioned slice with respect to the predicate $N < 0$. A vertex is made solid if it is ever executed and made bold if it gets traversed while computing the slice. The dotted vertices are not executed when the predicate is true

```
       P1                          P2                    P3


always @(clk or reset)       always @(clk)         always @(clk or flag)
  begin                         begin                   begin
   if (valid)                   reset = init;            start = flag;
       {                        if (valid)              end
           result = a+b;             {
       }                                flag = 1;
   else                              }
       {                        else
           result = a-b;             {
       }                                flag = 0;
                                     }
 end                          end
```

Figure 5.4. Example Verilog program. The three “always” blocks represent concurrent processes.

```
       P1                      P2                  P3


always @(clk or reset)   always @(clk)      always @(clk or flag)
 begin                     begin                 begin
   if (valid)                reset = init;         start = flag;
     {                       if (valid)          end
      result = a+b;           {
     }                           flag = 1;
                               }
 end                       end
```

Figure 5.5. The conditioned program, for the predicate (valid = true). Each process is a conditioned process.

```
        P1                      P2              P3

always @(clk or reset)   always @(clk)     always@(clk or flag)
 begin                    begin              begin
   if (valid)               reset = init;

      {                   end                end
       result = a+b;
      }

 end
```

Figure 5.6. The slice obtained by statically slicing the conditioned program.

antecedent_conditioned_slice ($P, G(a => X^{=n}C)$**)**
begin
$mark_{P_{-1}} = \emptyset$;
if $n \geq 0$
for every time step $0 \leq t \leq n$
begin
$mark_{P_t} = mark_{P_{t-1}} \coprod get\_conditioned\_slice(P, t, \langle a, i, V_h \rangle)$
end
$mark_P = mark_{P_n}$
S_a = prune($mark_P$)
S_a = get_static_slice (S_a, i, V_h)
return (S_a)
end
get_conditioned_slice ($P, t, < a, i, V_h >$**)**
begin
E_k = compute_fixpoint_expression (P, a)
for every process p in P do
begin
for every statement s in process p do
begin
sym_s = symbolic_expression($s, k + t$)
$mark_s$ = decision_procedure (sym_s , E_k)
end
end
return $mark_P$
end
compute_fixpoint_expression (P, a**)**
begin
$E_{(k+1)}$ = T
E_k = a
while (E_k != E_{k+1}) do
begin
for each process p in P do
for each statement $v = f(X)$ in process p
if (v is a variable in E_k)
begin
sym_v = symbolic_expression($v = f(X), (k - 1)$)
$E_{k-1} = E_{k-1} \vee sym_v$
end
$E_k = E_k \vee E_{k-1}$
$k = k - 1$
end
return (E_k)
end
prune(P**)**
begin
$S = P$
for every path ρ in P
for every sub-path ρ' in ρ until the endpoint
if every statement s in ρ' has ($mark_s$ == F)
$S = S - \rho'$
return (S)
end

Figure 5.7. Algorithm for antecedent conditioned slicing.

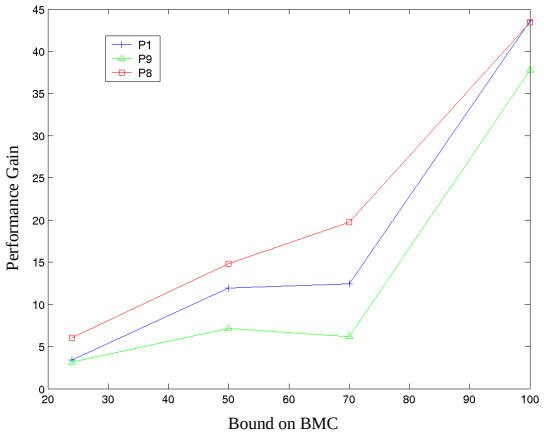


Figure 5.8. Graph showing the performance gain (speedup) of antecedent conditioned slicing for increasing bounds on BMC

Chapter 6

Processor Verification

6.1 Introduction

Formal verification of a pipelined microprocessor is a challenging and hitherto unsolved problem. Theorem proving approaches have produced an outstanding body of research related to the verification of complex processors (e.g., [172, 208]). Nonetheless, the drive for full automation is well-motivated, particularly in an industrial setting where simulation-based verification techniques remain the primary mechanism to validate modern industrial processors.

Model checking [58] algorithms, although automatic, suffer from high resource requirements and are prone to exhausting memory or time limits on more complex design components. As such, these techniques have not been applied to processor verification as much as deductive verification techniques. It is desirable to use abstraction techniques to reduce the complexity of the verification problem, to enable the use of automatic model checking algorithms.

In [225, 227], we introduced *antecedent conditioned slicing*. When applied to Register Transfer Level (RTL) designs described in Hardware Description Languages (HDLs), antecedent conditioned slicing shows significant performance gains.

The efficacy of *domain aware analysis* is completely evidenced by this work. Due to the analyses done at the high level, it is possible to identify and deploy different effective techniques for solving a very important, challenging problem.

We apply antecedent conditioned slicing to automatically decompose the pipelined microprocessor verification problem into simpler sub-problems that can be handled by automatic model checking engines. In the context of processor verification, this automatic abstraction technique provides a channel for tapping into the efficiency of these state-of-the-art lower level engines. Our technique involves an instruction-wise decomposition (instruction slicing).

An instruction's execution is represented by an LTL property, with the instruction opcode as the antecedent. We prune the HDL description of the processor using antecedent conditioned slicing. We pass the resulting antecedent conditioned slice through a model checker. We thus check if every instruction's operation is as specified. We also prove certain other lemmas to ensure that there is no interference from incorrect results of other instructions. We thereby verify the correctness of the interactions among the instructions of a pipelined processor.

The antecedent conditioned slicing algorithm provides a conduit to its application to processor verification. We thereby present a revised algorithm for instruction slicing. We also discuss in detail, the efficacy and expediency of applying antecedent conditioned slicing to instruction slicing for processor verification. This is a demonstration of how the domain awareness of the target domain, i.e processors benefits the mode of analysis. influences the mode of analysis. We provide an argument for establishing the correctness of pipelined processor verification using our approach. We present our experiments on an RTL implementation of the OR1200, an off-the-shelf pipelined microprocessor. We show verification results of all instruction classes of this processor using our technique. In a previous version of this work, we had used the SMV[177] model checker as the lower level engine to show the results of our technique.

In this work, we demonstrate the efficiency of our RT-level static analysis technique using several other Boolean level verification engines as the *decision procedures*.

The verification algorithms applied in the lower level engines include counterexample guided localization refinement [61], k-induction based SAT solvers [213], and Symbolic Trajectory Evaluation (STE)[36]. The tremendous performance gains of using our static analysis technique, as compared to the inability of the lower level engines to verify the original processor model in reasonable time, is testimony to the efficacy of high level static analysis. Our antecedent-conditioned slicing enables dramatic performance improvements to all of these algorithms. In many cases, it makes the difference between being able to achieve a conclusive result or not.

The principal contributions of this work are:

- We introduce an automatic decomposition technique for microprocessor verification using antecedent conditioned slicing. The decomposition can be used for splitting the processor verification problem space into more tractable problems.

- Our algorithm provides a channel to leverage the power of automatic Boolean level engines for processor verification. We have shown the benefits of using our RT-level static analysis technique synergistically with SAT, BDD and STE based engines, using different verification algorithms.

- We describe a verification technique that works with any generic processor design described in RTL, as opposed to a high-level model of the processor. This, as stated in [8] is a very relevant issue in contemporary hardware verification environments.

- We demonstrate an alternative notion of verification coverage, by verifying all the instructions in a processor, as opposed to traditional notions of checking a pipeline. We provide an argument

for pipeline correctness using this approach. In doing so, we provide a generic framework for verification of single instruction issue, multi-stage pipelined processors in RTL.

The organization of the chapter is as follows. Section 6.2 describes the the algorithm for instruction verification using antecedent conditioned slicing and gives a sketch of the proof methodology. Subsection 6.2.2 discusses how processor verification is an excellent application for the antecedent conditioned slicing technique. In Subsection 6.2.3, we provide an intuition for the correctness of our notion of pipelined processor verification. In Section 6.3, we discuss the OR1200 processor model. In the rest of Section 6.3 we detail the proof methodology for that processor and present the experimental results of verifying the OR1200 instruction classes using different Boolean level verification algorithms. Section 6.4 describes related work in this field and provides a qualitative comparison to relevant work. We discuss the limitations and applications of our technique and conclude with Section 6.5.

6.2 Processor Verification using Antecedent Conditioned Slicing

In pipelined processor verification, Burch and Dill's [42] pioneering technique reasons about the entire state of the pipeline using an abstraction function. Since the abstraction (flushing) function is very complex for any reasonably sized processor, there has been subsequent work in refinement of the abstraction function, in order to create more tractable parts of it.

A processor's microarchitecture is described in terms of its instruction set. It is therefore intuitive to reason about the behavior of individual instructions while verifying the processor. In contrast to Burch and Dill, we do not reason about the entire state of the pipeline, but about individual instructions. This approach was also used by Jhala and McMillan in [133]. Since every instruction is implemented deterministically, with finite resources, this is tantamount to verifying many small finite state machines.

We achieve this instruction-wise decomposition using antecedent conditioned slicing. In order to apply antecedent conditioned slicing to processor verification, we model an individual instruction's behavior as an LTL property whose antecedent corresponds to the opcode of the instruction being verified. An instruction's behavior is expressed as a property of the form $[G(a \Rightarrow X^{=n}c]$[1]. For a pipelined processor, the antecedent at every time step involves the stages of the pipeline that an instruction needs to pass through to get the required output. Antecedent conditioned slicing, therefore eliminates the portion of the system behavior that is irrelevant to the instruction. Only those portions of the pipeline that are affected during the lifetime of an instruction, are retained. The resulting single instruction machine abstraction is used to verify the LTL property. Evidently, this abstraction provides a much smaller and simpler system to the model checker. We check every instruction's behavior using the above approach. The property checks if the instruction being verified performs the specified operations with respect to its intended destination (target) registers.

However, in order to guarantee correctness of the pipeline processor, we also need to consider interaction between instructions. This is done by proving ancillary lemmas (expressed as temporal properties) that check (a) whether the pipeline control signals like stalls, freeze, data forwarding, program counter, *etc.*, function according to specification; (b) whether the instruction being verified does not modify non-target registers. This is to ensure the correctness of subsequent instructions in the pipe. We prove these lemmas once for the entire processor design, using a model checker. We discuss these lemmas in detail in Subsection 6.3.1.

[1]We also allow liveness properties in our property specification language. However, verification related properties are typically safety properties.

6.2.1 Algorithm for Instruction Verification

Let P be a Verilog program describing a pipelined processor microarchitecture, with pipeline depth $n+1$.

We outline the algorithm for verifying an instruction I of processor P using our technique in Figure 6.1. We compare it with the generic antecedent slicing algorithm in Figure 5.7, in order to show why instruction based slicing for processor verification is an excellent application for the antecedent conditioned slicing technique.

Let $h = [G(I \Rightarrow R)]$ be an LTL formula. Let $I = i_1 \wedge Xi_2 \wedge XXi_3 \ldots \wedge X^n i_n$ be the antecedent of h, where i_t represents the antecedent in cycle [2] t. R represents the expected result of executing instruction I, in terms of target register values.

Please note that the antecedent in the first cycle is the instruction word. The antecedent is iteratively refined in successive cycles by specifying control signals at each cycle [3].

In Figure 6.1, the procedure $insn\_check()$ computes the antecedent conditioned slice over the specified k time steps. It is similar to the $antecedent\_conditioned\_slice()$ procedure in Figure 5.7. For the first time step $t = 0$, the antecedent conditioned slice consists of the set of statements that would be executed when I holds. For any future time step t, the antecedent conditioned slice contains the set of

[2] In Verilog programs describing sequential hardware circuits, a clock is explicitly modeled in the design. Successive time steps are, therefore, according to the progression of this clock (cycles). We, therefore, use "time step" interchangeably with "cycle".

[3] We obtain the control signals in subsequent cycles from the counterexamples generated by a model checker. This is illustrated in Subsection 6.3.1.

single_insn_check (P : Verilog program, I : instruction, h : LTL property)
begin
  $h = [G(I \Rightarrow X^{=n} R)]$
  $insn\_check(P, h)$
end
insn_check (P, h)
begin
  $mark_{P_{-1}} = \phi$
  for every time step $t \geq 0$, while $t \leq n$
  begin
    $mark_{P_t} = mark_{P_{t-1}} \coprod get\_conditioned\_slice(P, t, \langle I, i, V_h \rangle)$
  end
  $mark_P = mark_{P_n}$
  S_a = prune($mark_P$)
  S_a = get_static_slice (S_a, i, V_h)
  return $model\_check(S_a, h)$
end
get_conditioned_slice ($P, t, < a, i, V_h >$**)**
begin
  for every process p in P do
  begin
    for every statement s in process p do
    begin
      sym_s = symbolic_expression(s, t)
      $mark_s$ = decision_procedure (sym_s , a)
    end
  end
  return $mark_P$
end
prune(P**)**
begin
  $S = P$
  for every path ρ in P
    for every sub-path ρ' in ρ until the endpoint
      if for every statement s in the ρ' ($mark_s$ == F)
        $S = P - \rho'$
  return (S)
end

Figure 6.1. Algorithm for verifying an instruction I using antecedent conditioned slicing.

statements that would be executed t time steps later, when I holds in $t = 0$.

The procedure $model\_check()$ shows the step where the antecedent conditioned slice is then passed through a model checker along with the property h. In case the property is disproven, a counterexample is returned.

6.2.2 Expediency of Applying Antecedent Conditioned Slicing to Processor Verification

When antecedent conditioned slicing is applied to instruction slicing for processors, some features are observed, that are advantageous with respect to efficiency and lower computational effort. The important considerations that provide performance benefits are as follows.

- The antecedent is the instruction (word), which is an input, [4] in the case of processor verification. This eliminates the call to $compute\_fixpoint\_expression()$, in the $get\_conditioned\_slice()$ procedure, that was a part of the generic algorithm. Since the antecedent is an input, it is not necessary to find the symbolic expression of the antecedent. This reduces the corresponding computational effort. This is a substantial reduction in the overall computational complexity of the algorithm. In theory, this step can be computationally intensive, and eliminating it increases the efficiency of the technique considerably.

- In instruction based slicing, an instruction's complete behavior, from the stage when it is fetched, to the stage when it is written back, is isolated. This provides an "independence" of every slice

[4] Sometimes the instruction word may not be an input, but a program stored in the RAM. The discussion still holds in that case.

from the other parts of the Verilog program, since the other parts represent behaviors of other instructions.

In order to determine the truth value of the antecedent at a given statement, the $decision\_procedure()$ in the generic algorithm could be a rewriting engine or a SAT solver that would evaluate the contradiction between two Boolean expressions. This could be a computationally intensive step that depends heavily on the efficiency of the decision making engine, especially if the two expressions are complicated. However, in the case of instruction-based slicing, the decision making procedure is relatively much simpler. This is due to the independent nature of the system behavior with respect to each instruction that makes the symbolic expressions for the Verilog program variables more tractable than for a generic Verilog program. Also, since the antecedent is an input, and not assigned within the program, the symbolic expression for the antecedent is not complicated. This makes the evaluation of an instruction's truth value at any point in the program significantly less complex than the generic case. Consequently, the cases where the decision procedure would return an X due to its inability to determine the truth of the antecedent, will be minimal.

- The antecedent (instruction), once it is true, does not change its truth value with respect to the property being checked. In other words, within the scope of the property being checked, namely, instruction behavior, the instruction does not toggle. The $prune()$ procedure in the instruction slicing algorithm is very simple in the case of instruction slicing. In the generic case, where the truth value of the antecedent is liable to change (toggle) all paths of the program have to be analyzed for toggling behavior. If toggling is observed, the entire path needs to be retained. Since in the case of processor designs, the truth value of the antecedent *does not change along a path*, the path traversal step can be optimized to be more efficient. If the antecedent is not true at a node (program statement) in the control flow graph, it can be assumed that along all subsequent paths

from that node until an endpoint, the antecedent will not be true. Due to this "independence" in system behavior according to an instruction, a large portion of the Verilog program can be sliced away, when considering a single instruction.

We have thus shown how the complexity of the generic antecedent conditioned slicing algorithm is greatly simplified when applied to instruction based slicing of processors. As a result, the algorithm can be implemented very efficiently for this application.

6.2.3 Correctness of our notion of pipelined processor verification

We present an argument to show that our notion of pipelined processor verification is correct and complete.

Correctness Theorem

Let P be a pipelined processor with n stages in its pipeline. Let $O(P)$ be the observed state of P, consisting of the register file, the memory, the non-programmer-visible (temporary) registers and the program counter at any cycle. $O(P)$ is therefore, a superset of the programmer visible state. Let $I_1, I_2, \ldots I_n$ be the instructions at any point of time in the pipeline. If all these instructions execute correctly, they update $O(P)$. Let ψ be the antecedent conditioned slicing operation that performs the instruction-wise decomposition of P. If $Q_i = \psi(P)$ for a given instruction, then let Q represent the set of all instruction slices $Q_1, Q_2 \ldots Q_k$, where k is the size of the instruction set of P. Let $Q_1, Q_2, \ldots Q_n$ be the instruction slices corresponding to an arbitrary sequence of instructions $I_1, I_2 \ldots I_n$.

Let $O(Q_n)$ be the observed state of Q_n. Let ξ be an execution operation of an instruction that updates the observed state of P, after executing a sequence of zero or more instructions. Then, the final observed

state after executing instructions $I_1 \ldots I_n$ in P's pipeline (shown by $\|$) is given by $O(P_n)$.

$$O(P_n) = \xi(O(P), (I_1 \parallel I_n))$$

If now, $Q_1, Q_2, \ldots Q_n$ are executed in sequence, the following operations represent the updated final state of Q.

$$O'(P_1) = \xi(O(P), Q_1)$$
$$O'(P_2) = \xi(O'(P_1), Q_2)$$
$$\vdots$$
$$O'(P_n) = \xi(O(P_{n-1}), Q_{n-1})$$

Then, we have to prove that $O(P_n) = O'(P_n)$.

Proof Intuition

The above theorem states that the instruction slices, when executed in the same sequence as the corresponding instructions in the original pipelined machine, will produce the same state as the original pipeline. In order for this theorem to be true, we need to prove the following.

- Each instruction slice $Q_1 \ldots Q_n$ executes correctly, *i.e* it *produces the same result* as $I_1 \ldots I_n$ respectively, with respect to updating the target registers.
- Each instruction slice $Q_1 \ldots Q_n$ *does not alter* the processor state incorrectly at any cycle during its execution, *i.e* the visible and non-visible registers do not get updated with a wrong value.

These two steps provide the necessary and sufficient conditions for the correctness of the pipelined processor using our technique. Therefore, for any given pipelined processor, we need to prove these two results.

6.3 Verification of the OR1200 processor

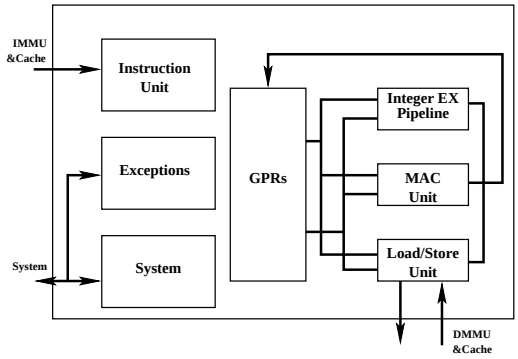


Figure 6.2. CPU block diagram of OR1200

As discussed in Subsection 6.2.3 we need to prove two results for proving the correctness of a pipelined processor. This proof is broken down into three parts. In the first part, we show that the current instruction is correct, *i.e* it updates the correct target register(s) with the functionally correct computation. In the second part, we prove that the control logic of the pipe *i.e* the flush, freeze, stall,[5] program counter and data forwarding (bypass) logic function according to specification. In the third part, we prove that the current instruction does not modify non-target registers in the register file (or non-target memory locations in the case of STORE instructions). We now detail our proof methodology, when applied to a pipelined processor microarchitecture.

We use the OpenRisc 1200, a publicly available processor for our experiments. The specification manual of the OR1200 is at [91] and the source code of its implementation in Verilog RTL can be

[5]This segregation of the control signals from the instruction's functionality is done for simplifying the problem. Since we are working with single instruction issue pipelines, this segregation is valid.

obtained from [188]. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. OR1200 is intended for embedded, portable and networking applications. Figure 6.2 shows the block diagram of the CPU of the OR1200 processor.

OR1200 implements 32 general-purpose 32-bit registers. Special purpose registers (SPRs) of all units are grouped under 32 groups. The load/store unit (LSU) transfers all data between the general purpose registers and the CPU's internal bus.

6.3.1 Proof methodology

We outline some details of the proof for an example instruction of the OR1200 processor. The `l.addc` (add with carry) instruction is specified in the instruction manual as follows.

```
rD[31:0] = rA[31:0] + rB[31:0] + SR[CY]
SR[CY] = carry
```

The instruction word is specified as given in [91].

Correctness Criterion for a Single Instruction Our correctness criterion is that given an LTL property whose antecedent corresponds to an instruction issue, and the consequent corresponds to the specified[6] output values, the LTL property should hold true on the processor's microarchitecture model.

Assumptions We disable the reads and writes from the debug unit to the special purpose registers (SPRS) unit. We also disable pending interrupts. We do this by ensuring that these values do not get asserted in every cycle.

[6]This refers to the specification manual of the processor, where each instruction's output values in target registers is given.

Instruction Behavior We describe an LTL property for the l.addc instruction. We obtain the desired antecedent by counterexample guided refinement. The instruction word is first given as an antecedent.

```
if ((icpu_dat_i[31:26]== 6'b 111000) &&
    (icpu_dat_i[9:8] == 2'b00)       &&
    (icpu_dat_i[3:0] == 4'b0001))
```

The add opcode is specified in the higher order bits of the instruction word, and the lower bits correspond to the opcode for the ALU. When passed to the model checker, this property generates a counterexample that shows a necessity to specify that a pipeline freeze or a pipeline flush has not been issued. Similarly, we also specify that a restore from exceptions is not necessary or there is no need to force fetch a delay slot. The antecedent is refined using the following conditions:

```
if (!rst && !flushpipe && !no_more_dslot && !rfe && if_freeze)
```

These two conditions capture the antecedent for the instruction fetch phase. We specify the antecedents for the next (decode) cycle, by stating the following:

```
wait(1);
if (!rst && !flushpipe && !id_freeze && !ex_freeze)
   wait(1);
```

The control conditions, as indicated by the counterexample, need to be specified every cycle. For instance, negating the signal `id_freeze` disables the freezing of the decode unit in the decode cycle.

When the decoded address reaches the register file, we latch the resulting two operands and carry in variables `a, b, c`. At the end of the writeback stage (5 cycles), we capture the result from the register file in `wbres` and compare it with the expected result. The consequent is written as follows:

```
assert addc: ((wbres == (a + b + c)));
```

Other Instruction Classes We verify all classes of instructions like Arithmetic Logic Unit (ALU) instructions, shift rotate (SHF/ROT) instructions, load store unit (LSU) instructions, branch and jump instructions, multiply and accumulate unit (MAC) instructions. The corresponding LTL properties can be found at [3]. For loads, we verify that the data is read from the correct address and loaded into the specified registers. For stores, we verify that the data specified in the source register is written to the correct target address. We do not explicitly verify the memory array itself.

Lemmas for Pipeline Correctness We now need to prove the lemmas that ensure pipeline correctness, according to the second part of our proof. We check that the program counter is incremented correctly for all instructions. We check the branch and jump instructions separately. Proving this result for the other instructions is trivial. We also prove auxiliary lemmas to ensure that the following control signals of the processor work correctly.

1. Pipleline flush signals: `flushpipe, extend_flush`
2. Freeze signals: `if_freeze, id_freeze, ex_freeze, wb_freeze, genpc_freeze`
3. Stall signals: `du_stall, mac_stall, lsu_stall, if_stall`
4. Data forwarding (bypass) logic: `ex_forw, wb_forw`

The third part of the proof is given by an additional lemma over the entire processor design that ensures that no incorrect values are written to other (programmer visible) registers in the register file. We disable some exceptions by not allowing illegal instructions. We also do not allow any stalls, freezes or flushing in the pipeline while proving this lemma.

There is only one write enable signal on the entire register file. There are two 32X32 dual-ported RAMs constituting the register file, one port is a read port and the other one is a write port. The `rf_we` signal decides when the register file is to be written. `rf_we` is asserted when an instruction writes back to the register file.

In order to prove that an instruction does not write incorrectly to any non-target register, we need to show that an instruction writes to the register file only during its write back stage, when it writes to its target registers.

```
assert lemma1:
     ((wbr == tr) &&
      ((reg_writeback_valid & rf_we) |
      (~reg_writeback_valid & ~rf_we)))
```

`reg_writeback_valid` represents the instructions that write back to the register file. For the instructions that write back to the register file, the write enable signal gets asserted in the cycle when the register being written (`wbr`) is the same as the target register (`tr`) in the instruction word. We also prove lemmas to ensure that the pipeline flush, freeze and stalls are performed correctly.

We have implemented our antecedent conditioned slicing algorithm in C++. We used our slicer to generate antecedent conditioned slices for each instruction. The slicing times taken by the slicer are to

the order of 300 seconds.

The antecedent conditioned slices for every instruction were then given to the model checker. We have used our methodology to verify all the instruction classes of the OR1200. All experiments were run on a 3 GHz Intel Pentium 4 processor with 1GB RAM. We used the SMV model checker [177] with the abstraction refinement (absref) option. This option utilizes SAT solvers and BDDs in combination to do the verification. The results of our experiments are shown in Table 6.1. On the original processor design, SMV, with the same options ran out of memory and did not finish the verification of the properties. We do not present these results, since all properties uniformly did not complete the verification process due to space limitations.

In Table 6.2, we present the results using different options in the SMV model checker. Each option corresponds to the implementation of a different algorithm. In order to analyze the efficacy and efficiency of our static analysis technique with different verification algorithms, we ran the antecedent conditioned slices using various Boolean level engines.

The column Localization shows the results of running a counterexample based localization refinement engine. Localization reduction with counterexample guided refinement was introduced in [61]. Localization reduction is an iterative technique that starts with an abstraction of the model under verification and tries to verify the property. When a counterexample is found, a reconstruction process is executed to determine if it is a valid one. If the counterexample is found to be spurious, the abstract model is refined to eliminate the possibility of this counterexample in the next iteration. This algorithm uses BDD-based lower level engines. These results have been obtained for runs without sifting or ordering the BDDs in these engines. The column k-induction corresponds to a safety property checking algorithm based on induction with depth, strengthened with a constraint that all states in a path be unique. This algorithm uses a SAT solver as a lower level engine, as described in [213].

Instruction Class	Instructions	SMV Time	Memory Usage
ALU	l.add	25.65s	23796KB
ALU	l.sub	24.70s	24018KB
ALU	l.addc	25.14s	25865KB
ALU	l.addi	21.60s	19658KB
ALU	l.addic	17.84s	16554KB
ALU	l.xor	24.84s	24831KB
ALU	l.and	23.28s	21727KB
ALU	l.or	24.01s	22761KB
BRANCH	l.bf	132.63s	44281KB
BRANCH	l.bnf	139.47s	46350KB
BRANCH	l.j	57.36s	31969KB
BRANCH	l.jr	59.64s	35177KB
BRANCH	l.jal	54.98s	31073KB
BRANCH	l.jalr	54.09s	30094KB
BRANCH	l.cmov	159.64s	49831KB
MAC	l.mul	25.28s	22801KB
MAC	l.mulu	26.63s	30004KB
COMPARE	l.sfeq	157.29s	51731KB
COMPARE	l.sfne	183.01s	53801KB
COMPARE	l.sfgt	194.43s	55352KB
COMPARE	l.sfge	206.39s	56904KB
COMPARE	l.sflt	201.10s	58146KB
COMPARE	l.sfle	275.97s	63112KB
LSU	l.ld	35.85s	29104KB
LSU	l.lws	33.91s	28873KB
LSU	l.lwz	35.27s	29567KB
LSU	l.sd	38.32s	30941KB
LSU	l.sw	39.30s	31365KB
SHF/ROT	l.sll	26.81s	23771KB
SHF/ROT	l.srl	27.83s	24865KB
SHF/ROT	l.sra	28.42s	30847KB
SHF/ROT	l.ror	27.93s	26919KB
SPRS	l.mfspr	226.97s	50696KB
SPRS	l.mtspr	212.27s	48627KB

Table 6.1. Time taken in seconds by SMV for verifying antecedent conditioned slices for all classes of instructions of OR1200 (all instructions not shown). Memory usage shown is total memory used for the entire verification operation (including both SAT and BDD phases). The unsliced, original design ran out of memory for all the properties.

Class	Instruction	Localization	k-induction
ALU	l.add	25.14s	16.71s
ALU	l.addc	17.84s	13.16s
BRANCH	l.j	57.36s	427.07s
BRANCH	l.jal	54.98s	396.73s
MAC	l.mul	25.28s	19.72s
MAC	l.mulu	26.63s	21.05s
COMPARE	l.sfne	183.01s	157.50s
COMPARE	l.sfgt	194.43s	179.81s
LSU	l.ld	35.85s	32.70s
LSU	l.sw	39.30s	37.00s
SHF/ROT	l.srl	27.83s	17.40s
SHF/ROT	l.sra	28.42s	23.26s

Table 6.2. Time taken in seconds after antecedent conditioned slicing, by engines implementing different Boolean level algorithms in SMV. The results are shown for two instructions per instruction class. The unsliced, original design ran out of memory for all the properties.

Instruction Class	Instruction	STE Unsliced	STE Sliced
ALU	l.add	549.12s	431.37s
ALU	l.addc	551.53s	459.41s
BRANCH	l.j	811.87s	539.90s
BRANCH	l.jal	804.41s	781.61s
MAC	l.mul	570.92s	420.32s
MAC	l.mulu	575.83s	395.54s
COMPARE	l.sfne	905.61s	641.59s
COMPARE	l.sfgt	899.75s	705.81s
LSU	l.ld	834.00s	597.42s
LSU	l.sw	800.50s	522.19s
SHF/ROT	l.srl	795.61s	505.89s
SHF/ROT	l.sra	801.37s	526.23s

Table 6.3. Time taken by an STE engine to verify the sliced and unsliced versions of the design

The original processor model, without preprocessing it with our technique, does not run to completion with any of these engines due to capacity issues.

We also used our static analysis technique to decompose the design before passing it through a Symbolic Trajectory Evaluation(STE)[36] engine as shown in Table 6.3. The STE engine completed the verification of the unsliced version of the design. The table shows that the benefits obtained over the STE algorithm are not as significant as over the other verification algorithms. This could be due to the similarity between the constraint based reduction heuristics used by the tool and our slicing algorithm. However, considerable manual effort was spent in specifying the input constraints of the design for the STE engine.

6.4 Related Work

There has been a significant amount of research in the field of pipeline processor verification as described in [7]. Most of these are variations of the Burch and Dill method [42].

Theorem proving techniques for processor verification have been widely researched. Sawada and Hunt [208] refined the Burch and Dill correctness criterion to verify a complex microprocessor with exceptions, stalls, interrupts etc using the ACL2 theorem prover. Velev and Bryant [241, 242] enhance the Burch and Dill method by efficiently using uninterpreted functions, equality etc. Hosabettu et al [125] use *completion functions* to represent the status of an instruction.

All these techniques try to verify complex processors with superscalar attributes. In order to demonstrate a correspondence between the implementation and the reference model, these techniques require complicated invariants whose construction requires expertise. In this work, we focus on completely automating the pipelined processor verification problem. We therefore show our results on a simple single instruction issue, multi-stage pipelined processor, that does not have interrupts, exceptions or other sophisticated features of a superscalar processor. In a sense, our work is orthogonal to the theorem proving efforts, as our drift is more toward automation, and less toward complex processor designs.

In recent times, Symbolic Trajectory Evaluation (STE) [36] techniques have been applied with considerable success to verify components of the Pentium 4 processor [28, 203, 211]. The STE and GSTE [132] techniques have been used to verify the floating point execution units in Intel's Pentium 4 processors. In [238], static program slicing was shown to benefit STE engines. Since our technique improves over static slicing and often creates smaller abstractions, we expect that our technique, when used in conjunction with STE engines, could yield similar or higher benefits.

Patankar et al [195] use an instruction based decomposition technique similar to ours in combination

with STE engines. However, this approach might not scale, since it involves creating detailed trajectory maps for all possible sequences of instruction flow in the Boolean domain.

Jhala and McMillan [133] used a compositional model checking approach for verifying pipelined processors. Their work is related to ours, since it is oriented toward processor verification on an instruction by instruction basis, as opposed to the Burch and Dill correctness criterion. However, their technique requires significant manual intervention to decompose the proof into manageable pieces using symmetry, temporal case splitting and abstract interpretation. Our abstractions, antecedent conditioned slices, are computed using a generic algorithm, making the decomposition process automatic.

An important difference between previous processor verification techniques and ours, is that we do not build our own processor models for the purposes of verification, but use a publicly available processor implemented in Verilog RTL.

6.5 Summary

We have introduced an abstraction technique that can make model checking of processors viable. The abstraction exploits the natural decomposition that instructions in a pipelined processor allow. Our technique can be viewed as an RT-level static analysis step that can be built on lower level Boolean engines, to increase their efficiency and scope. In its current form, our technique is most effective for single instruction issue, multi-stage pipeline processors, such as graphics and embedded processors.

Since we use abstractions created by syntactic transformations, the efficiency of our algorithm is program and property dependent. In the case of properties where the antecedent changes over a period, all future behavior of the program would need to be retained for each time step. In these cases, the antecedent conditioned slice would be quite large.

In that respect, processor verification is an excellent application for our abstractions. The antecedent, which is the instruction word in the single instruction machine, does not change through the duration of the property.

In order to specify the correct antecedent, we use the counterexample from a model checker to refine the antecedent. This could potentially be manually intensive. In future work, we plan to automate this step.

Our future work involves applying the technique to processors with more complex features, like out-of-order execution, register renaming, interrupts etc.

Chapter 7

Conclusions

We have presented a case for high level reasoning with the work in this dissertation. Since system verification is looming large as the grand challenge of the future decades, the atmosphere is ripe to accept a shift in methodology. While there has been a steady contribution of research in the field of formal verification algorithms, the sizes and complexity of the systems they target has grown by quantum bounds. The approaches that analyze a generic state space, therefore, are not able to measure up to this daunting challenge.

Adopting a novel practice that is verification centric is not reasonable or economically viable, with increasing time to market pressures and the inertial nature of legacy practices. This is probably the reason why many efficient techniques that advocate designs that are correct by construction, or design specifications in a functional language are not used in the industrial settings. Since our goal is to provide excellent engineering solutions to extant and futuristic problems, we propose techniques that can easily be incorporated into existing frameworks. This might not fulfill the archetypal scientific model, of dealing with theoretically elegant and well defined artifacts. For instance, the ill-defined formal semantics of HDLs have discouraged formal verification researchers from analyzing them. In order to develop engineering practices that are based on sound theoretical foundations, it is critical to bridge the gap between the research in verification, and its acceptability in practice. Toward this goal, attempts should be made

to revise the perception in the system design industry of formal verification as an esoteric science.

A viewpoint that could create high impact among design teams in industry is the proposal of methodologies that can work with the same languages and design hierarchy as they are used to. We hope that this is the first among many attempts to promote the advent of formal verification into mainstream system design.

Appendices

Bibliography

[1] Calypto Design Systems *http://www.calypto.com*.

[2] Digital Radio Mondiale *http://www.drm.org/*.

[3] OpenRISC 1200 Properties.
http://www.cerc.utexas.edu/∼shobha/OR1200_properties/.

[4] System C Reference Manual *http://homes.dsi.unimi.it/ pedersin/AD/SystemC_v201_LRM.pdf*.

[5] Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC '00: Proceedings of the 37th Conference on Design Automation (DAC'00)*, 2000.

[6] RTL Verification using Term Rewriting Systems. Technical Report 34, Computer Engineering Research Center, University of Texas at Austin, 2003.

[7] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144, pages 433–448, 2001.

[8] Mark Aagaard, Vlad Ciubotariu, Jason Higgins, and Farzad Khalvati. Combining equivalence verification and completion functions. In *Formal Methods in Computer-Aided Design (FMCAD 2004)*, 2004.

[9] Jacob A. Abraham, Vivekananda M. Vedula, and Daniel G. Saab. Verifying properties using sequential atpg. *International Test Conference*, pages 194–200, 2002.

[10] H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, 1991.

[11] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, 1990.

[12] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[13] M. Fahim Ali, A. Veneris, A. Smith, S. Safarpour, R. Drechsler, and M. Abadir. Debugging sequential circuits using boolean satisfiability. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 204–209, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1993.

[15] Demos Anastasakis, Robert Damiano, Hi-Keung Tony Ma, and Ted Stanion. A practical and efficient method for compare-point matching. In *Proceedings of the 39th conference on Design automation*, pages 305–310, 2002.

[16] Demosthenes Anastasakis, Lisa McIlwain, and Slawomir Pilarski. Efficient equivalence checking with partitions and hierarchical cut-points. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 539–542, New York, NY, USA, 2004. ACM Press.

[17] S. Antoy and J. Gannon. Using term rewriting to verify software. *IEEE Trans. Softw. Eng.*, 20(4):259–274, 1994.

[18] A. Appel and D. MacQueen. Standard ML of new jersey, 1991.

[19] Thomas Arts and Jurgen Giesl. Applying rewriting techniques to the verification of erlang processes. In *CSL*, pages 96–110, 1999.

[20] Nathaniel Ayewah, Sven Beyer, Nikhil Kikkeri, and Peter-M Seidel. Challenges in the formal verfication of complete state-of-the-art processors. In *IEEE International Conference on Computer Design (ICCD)*, 2005.

[21] R. Bagrodia. A distributed algorithm to implement the generalized alternative command of CSP. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 422–427, Washington, DC, 1986. IEEE Computer Society.

[22] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 29–40, 1993.

[23] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[24] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[25] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *POPL 2002*, pages 1–3, 2002.

[26] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 522–527, New York, NY, USA, 1998. ACM Press.

[27] Saddek Bensalem, Susanne Graf, and Yassine Lakhnech. Abstraction as the key for invariant verification. In *Symposium on Verification celebrating Zohar Manna's 64th Birthday*, 2003.

[28] B. Bentley. Validating the intel pentium 4 microprocessor. In *Proceedings of the 38th conference on Design automation*, pages 244–248, 2001.

[29] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37–68, 1999.

[30] T. Bolognesi and M. Caneve. Incremental development of a tool for equivalence verification, 1988.

[31] T. Bolognesi and S. A. Smolka. Fundamental results for the verification of observational equivalence: A survey. pages 165–178. H. Rudin and C. H. West, editors, 1987.

[32] A. D. Booth. A signed binary multiplication technique. In *Journal of Mechanics and Applied Mathematics*, pages 236–240, 1951.

[33] Robert S. Boyer and J. Strother Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[34] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[35] R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[36] R. E. Bryant, D. L. Beatty, and C. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th conference on ACM/IEEE design automation*, pages 397–402, 1991.

[37] R. E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Design Automation Conference*, pages 535–541, 1995.

[38] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Logic*, 2(1):93–134, 2001.

[39] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer Aided Verification*, pages 470–482, 1999.

[40] Gael N. Buckley and Abraham Silberschatz. An effective implementation for the generalized input-output construct of CSP. *Programming Languages and Systems*, 5(2):223–235, 1983.

[41] J. R. Burch. Using bdds to verify multipliers. In *Proceedings of the 28th conference on ACM/IEEE design automation conference*, pages 408–412. ACM Press, 1991.

[42] J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC '96: Proceedings of the 33rd Annual Conference on Design automation*, pages 552–557, 1996.

[43] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *International Conference on Computer-Aided Verification*, June 1994.

[44] Jerry R. Burch and Vigyan Singhal. Robust latch mapping for combinational equivalence checking. In *ICCAD*, pages 563–569, 1998.

[45] G. Cabodi, P. Camurati, and S. Quer. Symbolic exploration of large circuits with enhanced forward/backward traversals. In *Proceedings of the conference on European design automation*, pages 22–27, 1994.

[46] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.

[47] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: a first step towards migrating to client-server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[48] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance*, pages 424–433, 1994.

[49] G. Canfora, A. De Lucia, and M. C. Munro. An integrated environment for reuse reengineering C code. *Journal of Systems and Software*, 42:153–164, 1998.

[50] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log |V|)$. *Theoretical Computer Science*, 19:85–98, 1982.

[51] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt. Application of term rewriting techniques to hardware design verification. In *24th ACM/IEEE conference proceedings on Design automation conference*, pages 277–282, 1987.

[52] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, USA, 1988.

[53] Formality Equivalence Checker. Formal Verification products and solutions by Synopsys®. *http://www.synopsys.com/products/verification/verification.html*.

[54] Jiunn-Chern Chen and Yirng-An Chen. Equivalence checking of integer multipliers. In *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 169–174, New York, NY, USA, 2001. ACM Press.

[55] Yirng-An Chen and Randal E. Bryant. Phdd: an efficient graph representation for floating point circuit verification. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 2–7, Washington, DC, USA, 1997. IEEE Computer Society.

[56] C. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257, 1996.

[57] E. Clarke, O. Grumberg, M. Talupur, and D. Wang. High level verification of control intensive systems using predicate abstraction. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE.03)*, pages 55–64, 2003.

[58] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[59] E .M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[60] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 159–163, 1995.

[61] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[62] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[63] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, 2003.

[64] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[65] USB Source Code. http://www.opencores.org/pdownloads.cgi/list/usb.

[66] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 436–453, 2001.

[67] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[68] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. 2001.

[69] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 20–24, 2003.

[70] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[71] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[72] D. Cyrluk. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical Report SRI-CSL-93-12, Menlo Park, CA, 1993.

[73] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design, Theory, Practice, and Experience (Bad Herrenalb, Germany)*, volume 901, pages 203–222. Springer-Verlag, 1994.

[74] D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 135–146, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[75] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. In *LMS Journal of Computation and Mathematics*, volume 1, pages 148–200, December 1998.

[76] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[77] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[78] D. Dams, W. Hesse, and G. J. Holzmann. Abstracting C with abC. pages 515–520.

[79] S. Danicic, C. Fox, M. Harman, and R. Hierons. Consit: A conditioned program slicer. pages 216–226, 2000.

[80] M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, Mark Marman, Chris Fox, and M. P. Ward. Consus: A scalable approach to conditional slicing. In *IEEE Proceedings of the Working Conference on Reverse Engineering*, pages 181–189, 2002.

[81] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[82] DesignWare. Synopsys® products and solutions. *http://www.synopsys.com/products/ designware*.

[83] David L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. MIT Press, Cambridge, MA, USA, 1989.

[84] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[85] USB Specification Document. *http://www.usb.org/developers/docs/*.

[86] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of Second Irvine Software Symposium*, pages 131–145, 1992.

[87] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[88] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 189–202. ACM Press, 1998.

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

[90] M. J. Bertin et al. *Pisot and Salem Numbers*. user Verlag, Berlin, 1992.

[91] D. Lampret *et al*. Openrisc 1000 architecture manual. *http://www.cerc.utexas.edu/∼shobha/openrisc_arch3.pdf* 2003.

[92] J. Cl. Fernandez. *ALDEBARAN* : un Syst‘eme de V’erification par R’eduction de Processus Communicants. Th‘ese de Doctorat, Univ. Joseph Fourier-Grenoble I, France, July 1984.

[93] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[94] Kathi Fisler and Moshe Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design*, pages 115–132, 1998.

[95] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward conditioning: a new program specialisation technique and its application to program comprehension. In *9th IEEE International Workshop on Program Comprehesion*, pages 89–97, 2001.

[96] Z. Fu, Y. Mahajan, and S. Malik. zChaff Solver. In *http://www.princeton.edu/∼zchaff/zchaff.html*.

[97] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. In *IEEE Transactions on Software Engineering*, pages 751–761, 1991.

[98] S. J. Garland, J. V. Guttag, and J. A. Staunstrup. Verification of VLSI circuits using LP. In *The Fusion of Hardware Design and Verification*, pages 329–345, Glasgow, July 4–6 1988. IFIP WG 10.2, North Holland.

[99] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Conference on Automated Deduction*, pages 271–290, 2000.

[100] A. J. Camilleri M. J. C. Gordon and T. F. Melham. Hardware verification using higher-order logic. in d. borrione, editor, from hdl descriptions to guaranteed correct circuit designs, 1986.

[101] Michael J. C. Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 136–145, 1995.

[102] S. G. Govindaraju and D. L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 366–370. ACM Press, 1998.

[103] Shankar G. Govindaraju and David L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 115–119, 2000.

[104] Graf. A complete inference system for an algebra of regular acceptance models. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1986.

[105] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, 1997.

[106] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[107] VHDL Synthesis Interoperability Working Group. Ieee p1076.6/d2.01 draft standard for vhdl register transfer level synthesis.

[108] H. Anderson, P. Williams, and H. Hulgaard. Equivalence Checking of Combinational Circuits using Boolean Expression Diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7), 1999.

[109] M. Harman, R. M. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/post conditioned slicing. In *ICSM*, pages 138–147, 2001.

[110] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[111] N. Harman. Correctness and Verification of Hardware Systems using Maude. Technical report, University of Wales Swansea, 2000.

[112] H.Cho, G.Hachtel, E.Macii, B.Pleisser, and F.Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE TCAD*, 15(12):1465–1478, 1996.

[113] Matthew Hennessy. Acceptance trees. *Journal of the ACM*, 32(4):896–928, October 1985.

[114] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[115] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, 2003.

[116] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 440–451, 1998.

[117] H. Hermanns and M. Siegle. Computing bisimulations for stochastic process algebras using symbolic techniques. *Proceedings of 6th International PAPM Workshop*, pages 103–118, 1998.

[118] P. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[119] C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.

[120] J. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *X IFIP International Conference on VLSI (VLSI 99)*, Lisbon, Portugal, November 1999.

[121] G. Holzmann. The model checker spin, 1997.

[122] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, - 2000.

[123] Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations, Ed. by Zvi Kohavi and Azaria Paz, Academic Press*. 1971.

[124] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, 1988.

[125] R. Hosabettu, G. Gopalakrishnan, and M. K. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 521–537, 2000.

[126] http://www.eedesign.com/news/OEG20001017S0038.

[127] Shi-Yu Huang, Kwang-Ting Cheng, and Kuang-Chien Chen. Verifying sequential equivalence using atpg techniques. *ACM Trans. Des. Autom. Electron. Syst.*, pages 244–275, 2001.

[128] F. Huch. Verification of erlang programs using abstract interpretation and model checking. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261–272, 1999.

[129] W. A. Hunt. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas at Austin, 1985.

[130] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. pages 132–139, 1996.

[131] J. A. Abraham J. Baumgartner, A. Kuehlmann. Property checking via structural analysis. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 151–165, 2002.

[132] C.H. Seger J. Yang. Generalized symbolic trajectory evaluation - abstraction in action. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 70–87, 2002.

[133] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 396–410, 2001.

[134] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 2–6, 1995.

[135] Robert B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Computer Systems Laboratory, Stanford University, August 1999.

[136] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[137] K. Schneider and T. Kropf. Verifying Hardware Correctness by Combining Theorem Proving and Model Checking. Technical Report SFB358-C2-5/95, Institut für Rechnerentwurf und Fehlertoleranz, 1995.

[138] P. Kannellakis and S. Smolka. CCS Expressions. *Information and Computation*, 86:43–68, 1990.

[139] D. Kapur. Theorem proving support for hardware verification. In *Third Intl. Workshop on First-Order Theorem Proving (FTP 2000), St. Andrews, Scotland*, July 2000.

[140] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using powerlists, 1995.

[141] D. Kapur and M. Subramaniam. Extending decision procedures with induction schemes. In *Conference on Automated Deduction*, pages 324–345, 2000.

[142] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[143] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, pages 269–278, 1996.

[144] M. Kaufmann and J. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[145] M. Kaufmann. Personal Communication.

[146] M. Keim, M. Martin, B. Becker, R. Drechsler, and P. Molitor. Polynomial formal verification of multipliers. *VLSI Test Symposium*, pages 150–155, 1997.

[147] B. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell System Technology Journal*, 49(2):291–307, 1970.

[148] J. Klop. Term Rewriting Systems. In *In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors: Handbook of Logik in Computer Science, Oxford University Press*, volume 2, pages 1–116, 1992.

[149] Donald K. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[150] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[151] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[152] D. Kozen. Results on the propositional mu-calculus. In *Theoretical Computer Science*, volume 27, pages 333–354, 1998.

[153] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371, 2003.

[154] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 414–453, 1990.

[155] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[156] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer-Aided Verification 2003 (CAV 2003)*, pages 141–153, 2003.

[157] Leslie Lamport. *LaTeX: A document preparation system*. Addison-Wesley, 2nd edition, 1994.

[158] K. G. Larsen. *Context-dependent bisimulation between processes*. PhD thesis, 1986.

[159] Conformal^{TM} Logic Equivalence Checker (LEC). Verplex systems. *http://www.verplex.com/products/lec_broch*

[160] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for ctl model checking. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 76–81, 1996.

[161] J. Levitt and K. Olukotun. Verifying correct pipeline implementation for microprocessors. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 162–169, 1997.

[162] Jeremy R. Levitt and Kunle Olukotun. A Scalable Formal Verification Methodology for Pipelined Microprocessors. In *Design Automation Conference*, pages 558–563, 1996.

[163] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, 1985.

[164] J. Lind-Nielsen and H. Andersen. Stepwise CTL model checking of state/event systems. In *CAV'99: Computer Aided Verification* , 1999.

[165] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[166] Feng Lu, Madhu K. Iyer, Ganapathy Parthasarathy, Li-C. Wang, Kwang-Ting Cheng, and Kuang-Chien Chen. An efficient sequential sat solver with improved search strategies. In *DATE*, pages 1102–1107, 2005.

[167] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension*, page 9, 1996.

[168] F Mittelbach M Goosens and A Samarin. *The LaTeXCompanion*. Addison-Wesley, 1994.

[169] M. Kaufmann and D. Russinoff. Verification of Pipeline Circuits. In *ACL2 Workshop 2000 (proceedings are available as UTCS Technical Report TR-00-29)*, October 2000.

[170] Z. Manna, N. Bjorner, A. Browne, E. Y. Chang, M. Colon, L. de Alfaro, H. Devarajan, A. Kapur, J. Lee, H. Sipma, and T. E. Uribe. Step: The stanford temporal prover. In *TAPSOFT*, pages 793–794, 1995.

[171] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[172] Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement maps for efficient verification of processor models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1304–1309, Washington, DC, USA, 2005. IEEE Computer Society.

[173] IEEE 1364-2001 Standard Verilog Language Reference Manual.

[174] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. An equivalence checking method for c descriptions based on symbolic simulation with textual differences. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, pages 3315–3323, 2005.

[175] Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. In *Proceedings of Design Automation Conference*, pages 629–634, 1996.

[176] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 110–121, 1998.

[177] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[178] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence.*, pages 481–489, 1971.

[179] R. Milner. A calculus of communication systems. *Lecture Notes in Computer Science, Springer-Verlag*, 92, 1980.

[180] R. Milner. Communication and concurrency. In *Prentice-Hall International, London*. 1989.

[181] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[182] P. Mishra, N. Dutt, A. Nicolau, and H. Tomiyama. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 36, 2002.

[183] Dinos Moundanos, Jacob A. Abraham, and Yatin V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Trans. Comput.*, 47(1):2–14, 1998.

[184] M. Mutz. Using the hol prove assistant for proving the correctness of term rewriting rules reducing terms of sequential behavior. In *Proc. Workshop on Computer-Aided Verification*, pages 277–287, 1991.

[185] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, pages 435–449, 2000.

[186] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[187] E. R. Olderog and C. A. R. Hoare. Specification-oriented Semantics for Communicating Processes. Technical Report 8506, Christian–Albrechts–Universität Kiel, Olshausenstr. 40, D–2300 Kiel 1, West Germany, 1985.

[188] OPENCORES. http://www.opencores.org.

[189] K. J. Ottenstein and L.M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[190] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[191] A. Pardo and G. Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference*, 1998.

[192] Abelardo Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems*. PhD thesis, University of Colorado at Boulder, 1997.

[193] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981.

[194] D. Park. Concurrency and automata on infinite sequences, 1981.

[195] V.A. Patankar, A. Jain, and R.E. Bryant. Formal verification of an arm processor. In *Twelfth International Conference On VLSI Design*, pages 282–287, 1999.

[196] Doron A. Peled. *Software Reliability Methods*. Springer Verlag, New York City, New York, 2001.

[197] M. Pistore and D. Sangiorgi. A partition refinement algorithm for the - calculus. *Proceedings of CAV'96, Lecture Notes in Computer Science*, 1996.

[198] C. Pixley. Formal verification of commercial integrated circuits. In *IEEE Design and Test of Computers*, 2001.

[199] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 31– 2 1977. IEEE Computer Society Press.

[200] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[201] P.Wolfer and V.Lovinfosse. Verifying properties of large sets of processes with network invariants. 407:68–80, 1990.

[202] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *Proc. of the International Conference on Theorem*

Provers in Circuit Design: Theory, Practice and Experience, pages 129–156, Nijmegen, 1992. North-Holland.

[203] K. R. Kohatsu R. Kaivola. Proof engineering in the large: formal verification of pentium 4 floating-point divider. In *International Journal of Software Tools and Technology Transfer*, pages 323–334, 2003.

[204] John H. Reppy. Higher-order concurrency. Technical Report TR92-1285, 1992.

[205] T. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *Proceedings of the 2nd International Workshop on Software configuration management*, pages 46–55, 1989.

[206] M. Santoro and M. Horowitz. Spim: a pipelined 64 3 64-bit iterative multiplier. *IEEE J. Solid-State Circuits*, 24:487–493, April 1989.

[207] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. 10th International Computer Aided Verification Conference*, pages 135–146, 1998.

[208] J. Sawada and W. A. Hunt Jr. Results of the verification of a complex pipelined machine model. In *Conference on Correct Hardware Design and Verification Methods*, pages 313–316, 1999.

[209] J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. In J. Staunstrup and R. Sharp, editors, *International Workshop on Designing Correct Circuits*. North-Holland, IFIP Transactions A-5, 1992.

[210] D. A. Schmidt. Structure-preserving binary relations for program abstraction. pages 245–265, 2002.

[211] T. Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th conference on Design automation*, pages 1–6, 2003.

[212] R. Sharp and O. Rasmussen. Rewriting with constraints in t-ruby. In *Conference on Correct Hardware Design and Verification Methods*, pages 226–241, 1993.

[213] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.

[214] H. I. Shehata and M. D. Aagaard. A general decomposition strategy for verifying register renaming. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 234–237, 2004.

[215] X. Shen. Design and Verification of Speculative Processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[216] J. Sifakis. Property preserving homomorphisms of transition systems. *Lecture Notes in Computer Science*, 164:458–473, 1983.

[217] A. Silberschatz. Communication and synchronization in distributed systems, 1979.

[218] J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 98–109, 1998.

[219] Luc Smria, Renu Mehra, Barry Pangrle, Arjuna Ekanayake, Andrew Seawright, and Daniel Ng. Rtl c-based methodology for designing and verifying a multi-threaded processor. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 123–128, 2002.

[220] Michael Spivak. *The joy of TEX*. American Mathematical Society, Providence, R.I., 2nd edition, 1990.

[221] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, 1990.

[222] Colin Stirling. The joys of bisimulation. In *Mathematical Foundations of Computer Science*, pages 142–151, 1998.

[223] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *FME*, pages 43–71, 2001.

[224] J. Strother Moore. Personal Communication.

[225] S.Vasudevan, E.A.Emerson, and J.A.Abraham. Efficient model checking of hardware using conditioned slicing. In *4th Int. Workshop on Automated Verification of Critical Systems*, 2004.

[226] S.Vasudevan, E.A.Emerson, and J.A.Abraham. Efficient model checking of hardware using conditioned slicing. In *Preliminary Proceedings of 4th Int. Workshop on Automated Verification of Critical Systems*, 2004.

[227] S.Vasudevan, E.A.Emerson, and J.A.Abraham. Improved verification of hardware designs through antecedent conditioned slicing. In *International Journal of Software Tools and Technology Transfer*, July 2006.

[228] Naofumi Takagi, Hiroto Yasuura, and Shuzo Yajima. High-speed vlsi multiplication algorithm with a redundant binary addition tree. *IEEE Trans. Comput.*, 34(9):789–796, 1985.

[229] B. Tarski. A lattice-theoretical fixpoint theorem and its applications, 1955.

[230] F. Tip. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.

[231] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[232] Jiajin Tu. Viterbi decoder coprocessor in the drm application soc. *http://www.cerc.utexas.edu/∼shobha/master_*
2006.

[233] Jan L. A. van de Snepscheut. Synchronous communication between asynchronous components. *Information Processing Letters*, 13(3):127–130, 1981.

[234] Alf J. van der Poorten. Some problems of recurrent interest. Technical Report 81-0037, School of Mathematics and Physics, Macquarie University, North Ryde, Australia 2113, August 1981.

[235] C. van Eijk and J. Jess. Detection of equivalent state variables in finite state machine verification, 1995.

[236] S. Vasudevan, E. A. Emerson, and Jacob A. Abraham. Improved verification of hardware designs through antecedent conditioned slicing. *Technical Report UT-CERC-TR-JAA-05-1*, http://www.cerc.utexas.edu/
tr-acs.pdf, 2004.

[237] S. Vasudevan, V. Viswanath, R. Sumners, and J.A. Abraham. Automatic verification of arithmetic circuits in rtl using stepwise refinement of term rewriting systems. In *Accepted in IEEE Transactions on Computers*, To appear in Fall 2006.

[238] V. M. Vedula and J. A. Abraham. Taming the complexity of ste-based design verification using program slicing. *IEEE International High Level Design Validation and Test Workshop 2006*, 2006.

[239] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.

[240] V. M. Vedula, W. J. Townsend, and J. A. Abraham. Program slicing for ATPG-based property checking. *International Conference on VLSI Design*, pages 591–596, 2004.

[241] M. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 28, 2002.

[242] M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 112–117, 2000.

[243] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 26(6):107–119, 1991.

[244] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, pages 260–269, 1967.

[245] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979.

[246] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[247] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.

[248] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.

[249] H. Yu and J. A. Abraham. An Efficient 3-bit-scan Multiplier without Overlappong Bits, and its 64X64 Bit Implementation. In *Proceedings of 7th Asia and South Pacific Design Automation Conference*, January 2002.

[250] Z. Zhou, X. Song, F. Corella, E. Cerny, and M. Langevin. Description and verification of RTL designs using multiway decision graphs. In *Proceedings of the Conference on Hardware Description Languages*, 1995.

[251] Zheng Zhou and Wayne Burleson. Equivalence checking of datapaths based on canonical arithmetic expressions. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 546–551. ACM Press, 1995.

Vita

Shobha Vasudevan is a PhD candidate in the Electrical and Computer Engineering department at the University of Texas at Austin. She completed her M.S.E from the University of Texas at Austin in 2003, and her B.E from University of Mumbai in 2001. Her research interests are formal hardware verification, SoC verification, model checking, term rewriting systems, system level power management and correctness issues of multithreaded processors. She is now an Assistant Professor in the University of Illinois at Urbana Champaign.

Permanent address: 203, Devaarti
Narayanan Pathare Road
Mahim, Mumbai-400016

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.