Formal Equivalence Checking of Software Specifications vs. Hardware Implementations

by

Xiushan Feng

B.Eng. Harbin Institute of Technology, 1997

M.Eng. Institute of Computing Technology,
Chinese Academy of Sciences, 2000

M.Sc. University of British Columbia, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

January 2007

Abstract

Ever-growing complexity is forcing logic design to move above the register transfer level (RTL). For example, functional specifications are being written in software. These specifications are written for clarity, and are not optimized or intended for synthesis. Since the software is the target of functional validation, equivalence verification between the software specification and the RTL implementation is needed.

This thesis introduces new techniques to reduce the complexity of this verification and increase the capability of current verification techniques.

The first contribution improves the efficiency of sequential equivalence verification. I introduce a partitioned model checking approach using Annotated Control Flow Graphs (ACFG) to represent software specifications for sequential circuits. The approach partitions the software and hardware states based on the structure of the ACFG, and uses the flow and the edge annotations in the ACFG to guide the state-space exploration. Experimental results show that the new partitioned model checking approach runs faster than the standard global reachability analysis.

The second contribution increases the scalability of combinational equivalence verification between a high-level software specification and RTL. Unlike conventional RTL-to-gate combinational equivalence verification, there are fewer structural similarities between the two models, and it is harder to find equivalent points. Furthermore, each path through the software can compute a different result, and there are an exponential number of paths. I first adapt the concept of cutpoints from hardware verification and define the analogous concept of software cutpoints, then implement a proof-of-concept cutpoint approach in my verification tool for the TI C6x family of DSPs. Experimental results show large improvements in both runtime and memory usage. Next, I introduce cutpoints into the equivalence verification of software specifications vs. hardware implementations. I present a novel way to introduce cutpoints early, during the analysis of the software, rather than after a low-level hardware-equivalent has been generated, thereby avoiding the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. I evaluate this method on a challenge problem suggested by colleagues in industry. Experimental results show large improvements in runtime and memory usage due to the early cutpoint insertion.

Contents

List of Tables

List of Figures

List of Acronyms

ACFG	Annotated Control Flow Graph
BDD	Binary Decision Diagram
BLIF	Berkeley Logic Interchange Format
BMC	Bounded Model Checking
CFG	Control Flow Graph
CNF	Conjunctive Normal Form
CTL	Computation Tree Logic
CUDD	Colorado University Decision Diagram Package. The CUDD package provides functions to manipulate Binary Decision Diagrams
DSP	Digital Signal Processing/Processor
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GSTE	Generalized Symbolic Trajectory Evaluation
IA-32	Intel Architecture, 32-bit
ILD	Instruction Length Decoder
ISA	Instruction Set Architecture
LTL	Linear Temporal Logic
PLA	Programmable Logic Array
ROBDD	Reduced Ordered Binary Decision Diagram
RTL	Register Transfer Level

SAT	Boolean SATisfiability
SCC	Strongly Connected Component
SIB	Scale Index Base. Certain encodings of the ModR/M byte require a second addressing byte (this SIB byte)
SRT	A popular method for division in many processors. Named for its creators (Sweeny, Robertson, and Toker).
SVC	Stanford Validity Checker
TI	Texas Instruments
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VLIW	Very Long Instruction Word

Acknowledgments

First and foremost, I wish to thank my direct supervisor, Alan J. Hu. I enjoy his vast knowledge and owe him lots of gratitude for having a profound impact on this thesis. Throughout these years, he has given me much valuable advice on both research and life which I am very lucky to benefit from. Without him, this thesis would never have been completed.

I would also like to thank the other supervisory committee members, Eric Wohlstadter, Son Vuong, and Jin Yang for inspiration in many ways and valuable feedback on my research proposal and thesis. Thanks to my university examiners, Mark Greenstreet and Andre Ivanov, for helpful comments. Special thanks go to my external examiner, Robert B. Jones at Intel, for his suggestions on research examples and the enormous amount of time and effort that he put into my thesis.

I also owe my colleagues in the ISD lab thanks for helping me on this path and for making life at UBC more enjoyable. A special acknowledgment goes to Flavio M. De Paula for his solid industrial experience and help on questions about Verilog and VCS.

I would like to express my gratitude for my friends at UBC, particularly, Xiaodong Zhou and Jian Chen, for providing encouragement, exchanging knowledge, and serving as outlets for venting frustration during my graduate program.

Last but not least, I must acknowledge my family for supporting me all the time. Thank you to my grandfather for encouraging me to pursue the PhD degree. Thank you to my mom and dad for always being supportive. Special thanks go to my wife, Shu, and my daughter, Ashley, without whose love, encouragement and patience, I would not have finished this thesis.

Chapter 1

Introduction

1.1 Motivation

The traditional digital integrated circuit design process starts with a rough, high-level description that gives the functionality of the circuit. The functional description can be defined using timing charts, state transition graphs, hardware description languages (e.g., Verilog or VHDL), or recent high-level design languages (SystemC, SystemVerilog, SpecC, etc.). After that, this functional specification is transformed into more detailed descriptions step-by-step. The steps include the following: architecture analysis, logic design, logic synthesis, and physical design.

Architecture analysis is the first step in the design flow. In this step, the human designers explore different architectural requirements, such as performance, area, and power. After that, the designers generate a behavior-level design, which describes the functionality of each unit, the interconnections, and the resource allocations. Scheduling (the task of determining start times of operations subject to precedence) and resource binding (the task of determining which modules will execute these operations) are two important steps for architecture analysis. After architecture analysis finishes, logic design is undertaken. In this step, the Register Transfer Level (RTL) of the design is manually generated. This RTL design includes Boolean formulas and timing information.

Logic synthesis takes the output of logic design, i.e., RTL descriptions, and generates an optimized technology-specific network (netlist) of logic primitive modules (AND, OR, etc.) in a particular library. This step is often automatically executed by logic synthesis tools. Synthesis techniques of this step differ according to the nature of the circuit (combinational or sequential) or the intended implementation architecture (multilevel logic, PLA (Programmable Logic Array), or FPGA (Field Programmable Gate Array)).

Physical design is the last step in the design process. Design tools or humans generate the geometric patterns that define the physical circuit details that will be implemented on silicon. During this step, procedures such as floorplanning, placement, and routing are performed.

All the steps of the design process need equivalence verification checking to ensure that the implementation follows the specification and that there are no bugs introduced by manual design optimization or synthesis tools. For RTL or below, abundant research and tools are available to do these jobs. They have proven to be successful for many practical circuits. Considering the complexity of the design process and circuits, a new trend of design methodology is moving design and verification to higher levels. This is because with the increasing complexity of circuit design, functional verification is becoming more and more difficult. RTL simulation and verification have emerged as a major bottleneck of the design cycle. Increasingly, companies are writing high-level functional specifications in software, to allow much faster preliminary simulation. Such high-level functional specifications usually start in C or C-like languages. These specifications can be efficiently compiled into executable machine code to enable fast simulation [98]. In addition, these high-level specifications are flexible for modification and easy to maintain. Therefore, during the phases of architectural analysis and high-level design, the system designers can play with software specifications to achieve better performance or other goals by fast simulation. Just as the widespread adoption of logic synthesis generates the need for RTL-to-gate verification, analogously, the use of a high-level design methodology introduces a need for Software-to-RTL verification. Thus, it is necessary to use verification tools to determine the equivalence of software and hardware.

1.2 Contributions

Addressing formal equivalence checking of software specifications vs. hardware implementations, my thesis focuses on reducing the complexity of this verification and increasing the capability of current verification techniques. I have two major contributions: ACFG-based partitioned model checking and cutpoints for equivalence checking with software models.

1.2.1 ACFG-Based Partitioned Model Checking

In [38], I introduced a partitioned model checking approach using *Annotated Control Flow Graphs* (ACFG) to represent software specifications for sequential circuits. To model check the hardware implementation against the ACFG, the standard approach is to build a single product automaton combining the ACFG and the hardware model and then performing reachability analysis on the combined model. However, such an approach is prone to the state explosion problem.

My model checking solution, instead, is based on GSTE-style model checking proposed in [96]. The key idea is to partition the software states and the hardware states based on the structure of the ACFG, and use the flow and the annotation on edges in the ACFG to guide and tailor the state space exploration. More specifically, I compute a simulation relation $sim(e)$ for each edge e in the ACFG, such that the simulation relation tells us (1) what is the set of states the software model can be in on the particular edge e, and (2) for each such state, what is the corresponding set of possible states in the hardware model. The consequent on the edge is then checked against the simulation relation.

The experimental results show that in all test cases, the new partitioned model checking approach runs faster, sometimes much faster, than global reachability analysis.

1.2.2 Cutpoints for Embedded Software

Most work in my thesis is inspired by the success of RTL vs. gate-level combinational equivalence checking. A key technique behind this success is the idea of cutpoints [9, 12]: since the two combinational circuits are presumed to be structurally similar, there should be intermediate points in the two circuits that are logically equivalent. Heuristics search for such possibly equivalent intermediate points, and the tool first tries to prove such points equivalent. If successful, the equivalent logic behind the cutpoints is removed and replaced by a new primary input, thereby simplifying the verification problem. In general, the method is conservative (i.e., success proves equivalence, but failure doesn't prove inequivalence) because constraints on the cutpoint "inputs" are lost; various techniques re-introduce constraints to reduce this problem, e.g., [12].

We can see that cutpoints exploit the structural similarity of two combinational circuits, and therefore, simplify the verification. I believe that in verification of software specifications vs. hardware implementations, there exists a certain degree of similarity between the software model and the hardware implementation due to the similar functionality they have. Such similarities imply the existence of cutpoints. I give a definition of cutpoints for software: a cutpoint in software is some part of the program state at some point in one program, which is provably equal to some part of the program state at some point in another program.

Based on the above definition, I first apply software cutpoint theory to formal equivalence verification of embedded software to see the consequence of cutpoint insertion. In [36], I address several novel problems: how detailed will the cutpoint analysis be, how to find candidate cutpoints, and how to reduce false inequivalence. Then, I give solutions for the above questions and implemented a proof-of-concept cutpoint approach in my prototype verification tool for the Texas Instruments (TI) C6x family of very long instruction word (VLIW) digital signal processors (DSPs).

I have run experiments using several test cases which include compiler-optimized code and some expert hand-tuned DSP signal-processing routines from a TI-supplied library. Experimental results on these test cases show large improvements in runtime and memory usage over the previous state-of-the-art [60].

1.2.3 Early Cutpoint Insertion for Combinational Circuits

Building on the success of adapting the cutpoint idea to embedded software, I further introduce cutpoints into the formal equivalence verification of software specifications vs. hardware implementations [37].

Unlike the work for embedded DSP software, software paths in software specifications of hardware are more complicated. There are fewer similarities between the software specifications and hardware implementations — the same functionality is easy to describe sequentially in software, but it will be executed on highly parallel hardware. In addition, enumerating paths becomes difficult and suffers from exponential blow-up in the number of paths, and the corresponding equivalent points between the software and the hardware are not at all obvious. Fortunately, an analysis of the software can give us many high-level heuristics to guide the space exploration and abstraction. The first phase of my approach uses preliminary software analysis to unroll loops, merge paths as much as possible, and generate an unrolled graph. The second phase verifies the equivalence of the software and the hardware, while inserting cutpoints during the processing of the unrolled graph. The construction of the unrolled graph is linear in the size of the software (with all loop bodies instantiated). There is no exponential blow-up of path enumeration. In the formal equivalence checking phase, path conditions are computed on-the-fly. If some part of the program state at some point in the software is provably equal to some wire in the hardware, a new primary input is introduced in its place. This reduces the complexity of both hardware and software. If this process can be continued to the outputs of the software specification and hardware implementation, then equivalence has been formally verified.

I have run experiments on an industry-suggested challenge problem. Experimental results show that early cutpoint insertion can complete more instances than previous approaches. On those instances that the previous approaches can complete, the early cutpoint insertion achieves typically a 100x improvement in runtime and a 20x improvement in memory usage.

1.3 Thesis Outline

Chapter 2 provides a brief overview of background material. The next three chapters (Chapter 3, 4, and 5) present thesis contributions. Chapter 3 describes the ACFG-based partitioned model checking approach. In Chapter 4, I introduce a definition of cutpoints for software and apply it to embedded software. Chapter 5 explains the early cutpoint insertion approach for verifying software against combinational circuits. Chapter 6 contains my conclusions and suggestions for future research directions.

Chapter 2

Background

This chapter provides the background material for my research. This background is drawn from two areas: formal verification and program analysis. Formal verification is a broad area and has many subtopics. To limit the scope of this thesis, I present only the background that is related to the problems addressed in my thesis. As for program analysis, I apply simple, light-weight, static analysis techniques: dependence analysis and data-flow analysis. Therefore, I provide only a brief material to program analysis. This chapter gives the general background of my thesis; the specific related work for Chapters 3 – 5 is in each chapter.

2.1 Formal Verification

Formal verification is a technique to prove (or in some cases, disprove) the validity of an implementation with respect to a given specification using mathematical methods. The term *implementation* refers to the actual model of the system. *Specification*, on the other hand, refers to certain properties of the system. Such properties are described using formal methods (e.g., logic languages, state graphs, etc.). Compared with traditional simulation-based testing and other general reliability measures, formal verification is precise, well-defined, and assures a small probability of bugs slipping through unnoticed.

There are two major formal verification approaches: theorem proving and model checking.

In theorem proving, also known as deductive reasoning, the verification problem is described as a theorem in a formal theory. Given a set of axioms and a set of inference rules, the proof that the implementation realizes the specification is semi-automatically constructed and mechanically checked by a theorem prover (e.g., PVS [76], ACL2 [58], HOL [42], etc.) using deductive reasoning. With sufficient human ingenuity, any true (provable with the axiom system) theorem can eventually be proven. However, this approach is often time-consuming and is known to be labour-intensive, requiring considerable person-time to learn and use. In addition, there is no guarantee that the proving procedure will terminate.

Model checking, on the other hand, is less expressive, but can be fully automated and requires little time to learn. It is a technique to prove correctness of finite state concurrent systems. To be specific, the goal of model checking is to check whether a given model satisfies a given property using automatic decision procedures. Therefore, it guarantees termination. It has been successfully applied to digital sequential circuits and communication protocols. Since the focus of my thesis is on automatic formal verification, model checking is the more relevant approach to my work.

2.2 Model Checking Introduction

Model checking [24, 78] is a technique for automated verification of an implementation with respect to a specification. In order to perform model checking, three steps are performed: *modeling*, *specification*, and *verification*. In the step of modeling, we describe the system and convert the system into a formalism. Usually, the implementation of the system is described as a finite-state system (automaton). In the step of specification, we formulate properties as formulas in temporal logic (e.g., CTL, LTL, etc. described later) or some other formal description. At the last step, verification, efficient algorithms are used to traverse the model defined by the system and check if the specification holds or not. Commonly, a model-checking tool accepts an implementation and a property that the system is expected to satisfy. Then, the tool outputs "yes" if the given model satisfies the given properties. Otherwise, it generates a particular counterexample in the form of a sequence of states that violates the properties.

2.2.1 Modeling and Specification

Kripke Structure

First, I introduce how to do the modeling. The implementation, which is also referred to as a design, system, or model in this thesis, is usually some sort of a finite-state automaton, which is represented as a *Kripke structure* in the model checking domain.

A **Kripke structure** is a model used to give semantics (definitions of when a specified property holds) for modeling temporal logics. In the model checking domain, a Kripke structure is a graph having the states of the system as nodes and state transitions of the system as edges. It also contains labelings of the states in the structure with properties that hold in each state. The following is the formal definition.

Definition 2.1 (Kripke Structure) *Let AP be a non-empty finite set of atomic propositions that denotes the properties of individual states we are interested in. A Kripke structure is a four tuple $M = (S, s_{init}, R, L)$, where*

- *S is a finite set of states,*
- *$s_{init} \in S$ is the initial state,*
- *$R \subseteq S \times S$ is a transition relation,*

- $L : S \longrightarrow 2^{AP}$ *is a labeling function that attaches observations to the system. For a state* $s \in S$, $L(s)$ *is the atomic propositions that hold in state* s.

A *path* is an infinite sequence of states $\pi = \pi_0, \pi_1, \pi_2, \pi_3 ...$ where $(\pi_i \in S) \wedge ((\pi_i, \pi_{i+1}) \in R)$ for $i \geq 0$. π_j is *forward reachable* from π_i iff there exists a path to π_j and this path starts from π_i. π_i is *backward reachable* from π_j iff π_j is forward reachable from π_i.

The Kripke structure can be unwound into an infinite tree with the initial state s_{init} as the root, and each path in the tree represents a possible sequence of computation. Figure 2.1 shows a Kripke structure and its equivalent infinite computation tree. There are three states in this Kripke structure $\{s_0, s_1, s_2\}$. The initial state is s_0. The transition relations R is the set $\{(s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_2), (s_2, s_0)\}$. The labeling function $L = \{L(s_0) = \{P_0, P_2\}, L(s_1) = \{P_1\}, L(s_2) = \{P_1, P_2\}\}$, where P_i $(0 \leq i \leq 2)$ is an atomic proposition.

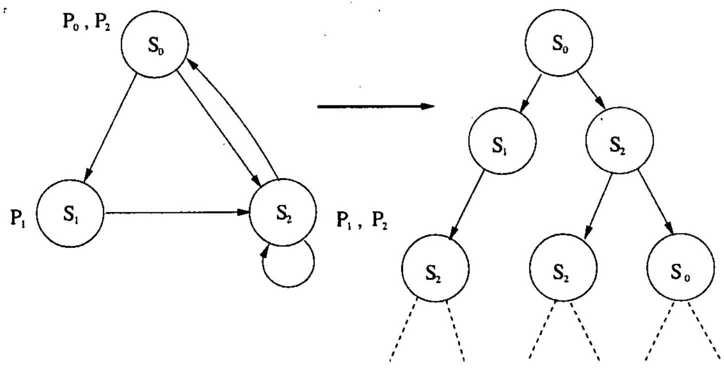


Figure 2.1: Kripke Structure and Computation Tree

Specification — Temporal Logic

Now that we have the model defined, we can introduce properties to characterize paths on a Kripke structure.

In order to capture the properties, temporal logic — a formalism used to describe how system states will change over time — is used as the specification language. In this thesis, I concentrate on *Computation Tree Logic* (CTL). It is a propositional, branching-time, temporal logic proposed by Clarke and Emerson in 1981 [24]. CTL, one of the most popular logics for practical model checking, uses atomic propositions as the basis, and formulas are constructed from logical operators, temporal operators and path quantifiers to make statements about the Kripke structure.

First, I will introduce the CTL* (pronounced "CTL-star") logic [32]. CTL is a restricted subset of CTL* where each of the temporal operators must be immediately preceded by a path quantifier. Informally, the operators are as follows.

- logical operators: $\vee, \neg$
- temporal operators:
  - Unary operators:
    - $\mathbf{X}\Phi$- Next (Φ has to hold at the next state)
    - $\mathbf{G}\Phi$- Globally (Φ has to hold on the entire subsequent path)
    - $\mathbf{F}\Phi$- Finally (Φ eventually has to hold on the subsequent path)
  - Binary operators:
    - $\Phi\mathbf{U}\Psi$- Until (Φ has to hold on the subsequent path until at some position Ψ holds)
- path quantifiers:
  - **A**: All (for all computation paths from the current state)
  - **E**: Exists (for at least one path from the current state)

The above operators and quantifiers can be used to express temporal properties of a Kripke structure. We are now ready to formally define the **syntax** of CTL*.

There are two types of formulas over the Kripke structure — *state formulas* and *path formulas*.

Definition 2.2 *If f is a CTL* state formula, it must be constructed by one of the following rules:*

- *f is an atomic proposition.*
- *$f = \neg g$, where g is a state formula.*
- *$f = g \vee h$, where g, h are state formulas.*
- *$f = \boldsymbol{A}p$, where p is a path formula.*
- *$f = \boldsymbol{E}p$, where p is a path formula.*

Definition 2.3 *If p is a CTL* path formula, it must be constructed by one of the following rules:*

- *p is a state formula.*
- *$p = \neg q$, where q is a path formula.*
- *$p = q \vee r$, where q, r are path formulas.*
- *$p = \boldsymbol{X}q$, where q is a path formula.*
- *$p = \boldsymbol{G}q$, where q is a path formula.*
- *$p = \boldsymbol{F}q$, where q is a path formula.*
- *$p = q\boldsymbol{U}r$, where q, r are path formulas.*

CTL* is the set of finite-length state formulas that can be constructed by applying the above rules. Here are two examples of CTL* formulas.[1]

- **AG**(`Req` $\Rightarrow$ **AF**`Ack`): if a `Req` (*Request*) occurs, then it will be eventually acknowledged (`Ack`). This is a *liveness property.* A liveness property declares

[1]In fact, these two formulas are also CTL formulas.

what should eventually happen. A simple counterexample to liveness properties is a path to a loop that does not contain the desired state, as such a loop represents an infinite path that never reaches the specified state.

- $\mathbf{AG}(\neg(pc_1 = CR_1) \vee \neg(pc_2 = CR_2))$: processes (pc_1, pc_2) will never be in their critical section (CR_1, CR_2) simultaneously. This is a *safety property* for a concurrent system. A safety property declares that something bad will never happen (or equivalently, what should always happen). Each violation of a safety property can be observed by looking at a finite history of the system behavior.

Before giving the semantics of CTL*, I will define the meaning of the $\models$ relation.

Definition 2.4 *If f is a CTL* state formula, and M is a Kripke structure, the formula $M, s \models f$ denotes that f holds at state s in the Kripke structure M.*

Definition 2.5 *If f is a CTL* path formula, M is a Kripke structure, and π is a path, the formula $M, \pi \models f$ denotes that f holds along path π in the Kripke structure M.*

Now, we are ready to define the semantics of CTL*. Definition 2.6 gives the formal **semantics** of CTL* over the Kripke structure.

Definition 2.6 *The relation $\models$ is defined inductively as follows. In the following definitions, s is a state, M is a Kripke structure, f_1, f_2 are state formulas, p_1, p_2 are path formulas, π^n is the suffix of the path π that begins at π_n.*

- *If $f_1 \in AP$, then $M, s \models f_1 \Leftrightarrow f_1 \in L(s)$.*

- $M, s \models \neg f_1 \Leftrightarrow M, s \not\models f_1$.

- $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1$ *or* $M, s \models f_2$.

- $M, s \models A(p_1) \Leftrightarrow$ *for all paths* π *starting with* s*, such that* $M, \pi \models p_1$.
- $M, s \models E(p_1) \Leftrightarrow$ *there exists a path* π *starting with* s*, such that* $M, \pi \models p_1$.
- $M, \pi \models f_1 \Leftrightarrow s$ *is the first state of* π *and* $M, s \models f_1$. *i.e., each state formula is also a path formula.*
- $M, \pi \models \neg p_1 \Leftrightarrow M, \pi \not\models p_1$.
- $M, \pi \models p_1 \vee p_2 \Leftrightarrow M, \pi \models p_1$ *or* $M, \pi \models p_2$.
- $M, \pi \models \boldsymbol{X} p_1 \Leftrightarrow M, \pi^1 \models p_1$.
- $M, \pi \models \boldsymbol{G} p_1 \Leftrightarrow$ *for all* $k \geq 0$, $M, \pi^k \models p_1$.
- $M, \pi \models \boldsymbol{F} p_1 \Leftrightarrow$ *there exists a* $k \geq 0$, $M, \pi^k \models p_1$.
- $M, \pi \models p_1 \boldsymbol{U} p_2 \Leftrightarrow$ *there exists a* $k \geq 0$ *such that* $M, \pi^k \models p_2$ *and for all* $0 \leq j < k$, $M, \pi^j \models p_1$.

After such definitions, the *model checking problem* is reduced to: given an implementation represented by a Kripke structure M, and a specified property ϕ in a certain temporal logic, determine whether $M, s_{init} \models \phi$.

As I said at the beginning of this section, CTL is more popular in the model checking domain due to its efficiency. Unlike CTL*, in which a formula can be composed of arbitrary combinations of temporal operators and path quantifiers, the logic CTL, on the other hand, given the constraint that each of the temporal operators must be immediately preceded by a path quantifier, ends up with only 8 basic operators: **AX** and **EX**; **AG** and **EG**; **AF** and **EF**; **AU** and **EU**. It is proven that the model checking problem for CTL* is in PSPACE and can be solved in time $2^{O(|\phi|)} \times O(|S| + |R|)$ [33], which is exponential in the size of the property's formula. In contrast with CTL*, the model checking problem for CTL is P-hard and can be solved in time $O(|\phi| \times (|S| + |R|))$, which is linear in the size of the Kripke structure and is linear in the size of the property's formula. This exponential time vs. linear time difference is due to CTL*'s arbitrary combinations of temporal operators and path quantifiers, which introduces more expressive power but more expensive computations as well.

There is another temporal logic — linear temporal logic (LTL [77], proposed by Pnueli) which is also frequently used in the model checking. It is also a subset of CTL*. Unlike CTL, the formulas of LTL don't have path quantifiers. Instead, LTL formulas are global, which means that every formula of LTL implicitly has the path quantifier **A** before it. Therefore, the holding of an LTL formula means the formula holds on all paths. It is impossible to say "there exists a path such that...". The complexity of LTL model checking is the same as that of CTL*, however, LTL has less expressive power. Figure 2.2 shows the relationship of the expressive powers of CTL*, CTL, and LTL. For example, there is no CTL formula that is equivalent to the LTL formula FGp (or $A(FGp)$ if we add the path quantifier **A** to avoid confusion). Likewise, there is no LTL formula that is equivalent to the CTL formula $AG(EFp)$. The disjunction $A(FGp) \vee AG(EFp)$ is a CTL* formula that is not expressible in either CTL or LTL.

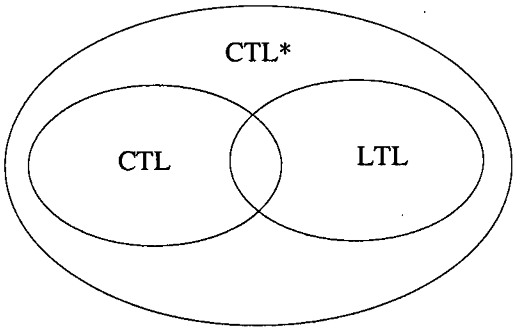


Figure 2.2: Expressive Powers of Three Logics

2.2.2 Verification Algorithms

Fixpoint Characterization

After we have defined the model (e.g., Kripke structure) and the specified properties (e.g., CTL*, CTL, or LTL formulas) on this model, the model checking algorithm will exhaustively search the state space of the model to determine the truth of the specification. For example, we want to check whether EGf holds at the initial state, i.e., we want to check whether there exists an infinite path (starting from the initial state), along which f is always true at all future states. In order to do that, we need to compute the set of states that f holds at the current state and EGf holds for some successors (f holds at the first two states of the infinite path), i.e., $Q_1 = f \wedge EXEGf$. For the successors in Q_1, we also need to check whether EGf holds or not for their successors. Then, we keep doing the same computation, $Q_2 = f \wedge EXEG(Q_1)$, etc., until we reach a set Q_{i+1} where $Q_{i+1} = Q_i$. We call this a *fixpoint*. If the initial state is inside the fixpoint, we have proven that EGf holds at the initial state. After some preliminary definitions [31], I will show how to use the fixpoint characterization for CTL operators.

Definition 2.7 (Predicate) *Given a Kripke structure $M = (S, s_{init}, R, L)$ and $x \in 2^S$, the set of states x defines a predicate PRED(x) on S. A state s satisfies $PRED(x)$ iff $s \in x$. A set of predicates comes with a natural partial order due to set inclusion/implication.*

For example, a state formula ϕ in CTL is a predicate. If x is the set of states satisfying ϕ, then $M, s \models \phi \Leftrightarrow s \in x$, where M is the given Kripke structure.

Definition 2.8 (Predicate Transformer) *A predicate transformer τ is a function that maps a predicate to another predicate. i.e, τ: $PRED(2^S) \rightarrow PRED(2^S)$. The predicate transformer τ is said to be monotonic, if $P \Rightarrow Q$ implies $\tau(P) \Rightarrow \tau(Q)$.*

```
function lfp($\tau$)
$Q := False$;
$Q' = \tau(Q)$;
while ($Q \neq Q'$) do
    $Q := Q'$;
    $Q' = \tau(Q)$;
end while
return ($Q$);
```

Figure 2.3: Algorithm to Compute the Least Fixpoint

For example, the operator **EX** in CTL is a predicate transformer that maps ϕ to $\mathbf{EX}\phi$, where ϕ is a CTL formula.

Definition 2.9 (Fixpoint) *A fixpoint of a predicate transformer τ is a predicate Q such that $\tau(Q) = Q$. i.e, if a predicate Q doesn't change by applying the predicate transformer τ, the predicate Q is a fixpoint of τ. The theorem of Tarski-Knaster [91] ensures that a monotonic function τ always has a least fixpoint ($\mathbf{lfp}Z.\tau(Z) = \bigwedge\{Q : \tau(Q) = Q\}$), and a greatest fixpoint ($\mathbf{gfp}Z.\tau(Z) = \bigvee\{Q : \tau(Q) = Q\}$).*

Figure 2.3 lists the algorithm for computing the least fixpoint for the predicate transformer τ.[2] For finite-state model checking, the algorithm terminates due to the fact that the model is a finite state system.

As I have mentioned, given a Kripke structure $M = (S, s_{init}, R, L)$ and a CTL formula ϕ, the model checking algorithm checks whether $M, s_{init} \models \phi$. This algorithm is equivalent finding the fixpoint Q for the predicate transformer corresponding to ϕ, and then checking whether s_{init} satisfies Q to ensure that the formula ϕ holds at the initial state s_{init}.

[2]Unlike the algorithm that computes the least fixpoint, the algorithm for computing the greatest fixpoint gives the initial value of Q an assignment *True* instead of *False* in line 2 of Figure 2.3. The other lines are the same. In practical model checking problems, τ is always monotonic.

- $p\mathbf{AU}q = \mathbf{lfp}\ Z[q \vee (p \wedge \mathbf{AX}Z)]$
- $p\mathbf{EU}q = \mathbf{lfp}\ Z[q \vee (p \wedge \mathbf{EX}Z)]$
- $\mathbf{AF}p = \mathbf{lfp}\ Z[p \vee \mathbf{AX}Z]$
- $\mathbf{EF}p = \mathbf{lfp}\ Z[p \vee \mathbf{EX}Z]$
- $\mathbf{AG}p = \mathbf{gfp}\ Z[p \wedge \mathbf{AX}Z]$
- $\mathbf{EG}p = \mathbf{gfp}\ Z[p \wedge \mathbf{EX}Z]$

Figure 2.4: Fixpoint Characterization of CTL Operators

Each of the CTL operators can be elegantly characterized as the least or greatest fixpoint of an appropriate predicate transformer, as shown in Figure 2.4 [31]. Therefore, the problem of finding the set of states that satisfies a CTL formula for a given Kripke structure is the same as the problem of finding a fixpoint for a predicate transformer using the algorithm in Figure 2.3. The computations for **EX** and **AX** in Figure 2.4 are as follows:

- $M, s \models \mathbf{EX}p \Leftrightarrow s \in \{s : \exists s'.R(s,s') \wedge (M,s' \models p)\}$, or equivalently
  $EXp = PRED(\{s : \exists s'.R(s,s') \wedge (M,s' \models p)\})$
- $M, s \models \mathbf{AX}p \Leftrightarrow s \in \{s : \forall s'.R(s,s') \wedge (M,s' \models p)\}$, or equivalently
  $AXp = PRED(\{s : \forall s'.R(s,s') \wedge (M,s' \models p)\})$

As an example, consider the **EG** operator. According to Figure 2.4, $\mathbf{EG}p = \mathbf{gfp}Z[p \wedge \mathbf{EX}Z]$. We need to compute the greatest fixpoint for **EG** using the algorithm from Figure 2.3, except we start Q_0 with an initial assignment $True$. The sequence of computations is:

- $Q_0 = True$
- $Q_1 = p \wedge \mathbf{EX}(Q_0) = p$
- $Q_2 = p \wedge \mathbf{EX}(Q_1) = p \wedge \mathbf{EX}(p)$

- $Q_3 = p \wedge \mathbf{EX}(Q_2) = p \wedge \mathbf{EX}(p \wedge \mathbf{EX}(p))$
- ...
- until $Q_{i+1} = Q_i$

When we reach this fixpoint Q_i, every state that satisfies Q_i has a successor that satisfies p in the Kripke structure. In other words, for every state that satisfies Q_i, there is an infinite path starting from the state and p always holds on the path.

Explicit Model Checking Algorithm

A direct implementation of the fixpoint characterization is explicit state enumerating, i.e., we use some data structure to explicitly store states and use graph traversal algorithms to find the states which satisfy the formulas. I will give a short description for such an algorithm in this section.

Such an algorithm is presented as three steps below. Just as in graph colouring, the algorithm marks the set of states where each CTL subformula holds.

First, we can rewrite the formula ϕ to ϕ_{trans} which contains only atomic operators like $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$. The other CTL operators can be rewritten as follows:

- $\mathbf{AX}p \Leftrightarrow \neg\mathbf{EX}\neg p$
- $\mathbf{AG}p \Leftrightarrow \neg\mathbf{EF}\neg p$
- $\mathbf{AF}p \Leftrightarrow \neg\mathbf{EG}\neg p$
- $\mathbf{EF}p \Leftrightarrow \mathit{true}\ \mathbf{EU}\neg p$
- $p\mathbf{AU}q \Leftrightarrow \neg(\neg p\ \mathbf{EU}\neg(p \vee q) \vee \mathbf{EG}\neg q)$

Second, for each subformula ϕ_{sub} of ϕ_{trans}, if ϕ_{sub} is an atomic proposition or a formula which has already been handled, the algorithm marks the set of states that satisfies the subformula ϕ_{sub}; if ϕ_{sub} is a formula which hasn't been handled, the algorithm recursively handles the subformula ϕ_{sub}. This step will terminate when the whole formula ϕ_{trans} has been handled.

Third, the algorithm checks whether the initial state is inside the final set of satisfied states, i.e., whether the formula ϕ holds at the initial state s_{init}.

I take the **EU** operator as an example to explain how the second step works (the handling of the logic operators is straightforward according to their meanings, and handling of the atomic propositions is just a simple searching and marking algorithm).

The input is: a set of states S_p that satisfies formula p and a set of states S_q that satisfies the formula q (i.e., assume p and q have been handled). We need the set of states where p **EU** q holds.

```
function EU($S_p$,$S_q$)
mark states $S_q$;
while new state is found do
  if there is a state in $S_p$ that has some successor state marked then
    mark it;
  end if
end while
return all marked states;
```

Figure 2.5: Algorithm to Compute **EU** Operator

Figure 2.5 shows the algorithm, which starts from S_q, and performs backward reachability using only states in S_p. It terminates because it is a monotonic function in the finite state space of the Kripke structure. We can see that the algorithm makes progress of at least one state per iteration, and for each iteration, it visits each state and edge at most once. Therefore, the algorithm is quadratic in the size of the Kripke structure, i.e., the complexity of the simple depth-first algorithm for the **EU** operator is $O(|S| \times (|S| + |R|))$, where $|S|$ is the size of the state space and $|R|$ is the number of edges in the Kripke structure. A more efficient algorithm is realized by remembering visited states and using backward breadth-first searching. Such an algorithm avoids visiting any state twice. The improved algorithm has complexity $O(|S| + |R|)$.

Similarly, we can introduce algorithms for the **EX** and **EG** operators. The algorithm for **EX**p is easy. It simply marks all predecessors of states S_p. An efficient algorithm for **EG**p is a little bit complicated. It needs to find an infinite path on which p always holds. In order to do that, we first restrict the Kripke structure to states satisfying p (i.e., remove all the other states and edges), then identify the maximal non-trivial Strongly Connected Components (SCC).[3] On such SCCs, **EG**p holds. By doing so, our problem is reduced to finding a path from the initial state to such SCCs on which p always holds (such a path is finite!). We can use a breath-first searching on the simplified Kripke structure. This algorithm has the same complexity as the efficient algorithm for the **EU** operator.

Therfore, the complexity of the model checking algorithm for a CTL formula ϕ is $O(|\phi| \times (|S| + |R|))$, i.e., it is linear in the size of the formula, and is linear in the size of the Kripke structure.

Although the CTL model checking algorithm is linear in the size of the Kripke structure, the size of the Kripke structure is exponential in the number of variables and concurrent components. For example, given a concurrent system with n components and m local states inside each component, the Kripke structure for such a system will have m^n global states! Such an exponential state space may introduce the biggest limitation of model checking, which is called the *state explosion problem*. With this state explosion, explicitly representing states often turns out to be impractical. In order to attack this problem, techniques, such as Boolean algebra, are used to implicitly represent sets of states and transition relations. By using Boolean formulas, the size of states that model checking can handle is greatly increased. In the next section, I will present such an approach.

[3]A component is strongly connected iff every node can be reached from every other node. Non-trivial means this SCC has ≥ 2 states or has one state with a self-loop

2.3 Symbolic Model Checking

Symbolic Model Checking usually uses BDDs, SAT, or mixed BDDs and SAT to prove the validity or to represent states and transition relations. In this section, I will introduce BDDs and SAT first. After that, I will explain the symbolic model checking approach by using an example.

2.3.1 BDDs and SAT

BDDs

In 1986, Bryant [14] proposed Reduced Ordered Binary Decision Diagrams (ROBDDs, or BDDs for short) by imposing restrictions on the representation first introduced by Lee [64] and Akers [2]. BDDs are efficient representations for Boolean formulas due to their compactness and canonicity. Canonicity is particularly useful: given equivalent Boolean formulas, BDDs for these formulas will be identical (for the same BDD variable ordering, described later). Therefore, the equivalence of two Boolean formulas can be reduced to comparisons of BDDs which can be checked in constant time. This canonicity, therefore, allows substantial subformula sharing, often resulting in the compactness of BDDs.

BDDs originate from ordered binary decision trees. Figure 2.6 is an example of an ordered binary decision tree for the Boolean function $f(x, y, z) = (xy+xz+yz)$. The variable ordering is given as $x < y < z$.

In Figure 2.6, each non-terminal vertex is labeled by a variable name x, y, z and has two children: one child is the case that the variable is assigned to the value 0 (0-arc, dashed lines), the other child is the case that the variable is assigned to the value 1 (1-arc, solid lines). Each terminal vertex is labeled 0 or 1 which is determined by the function f.

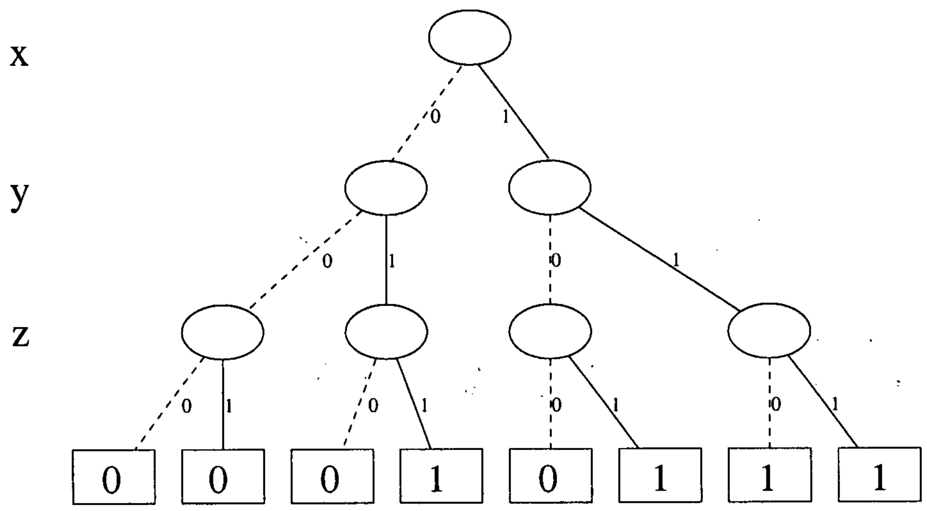
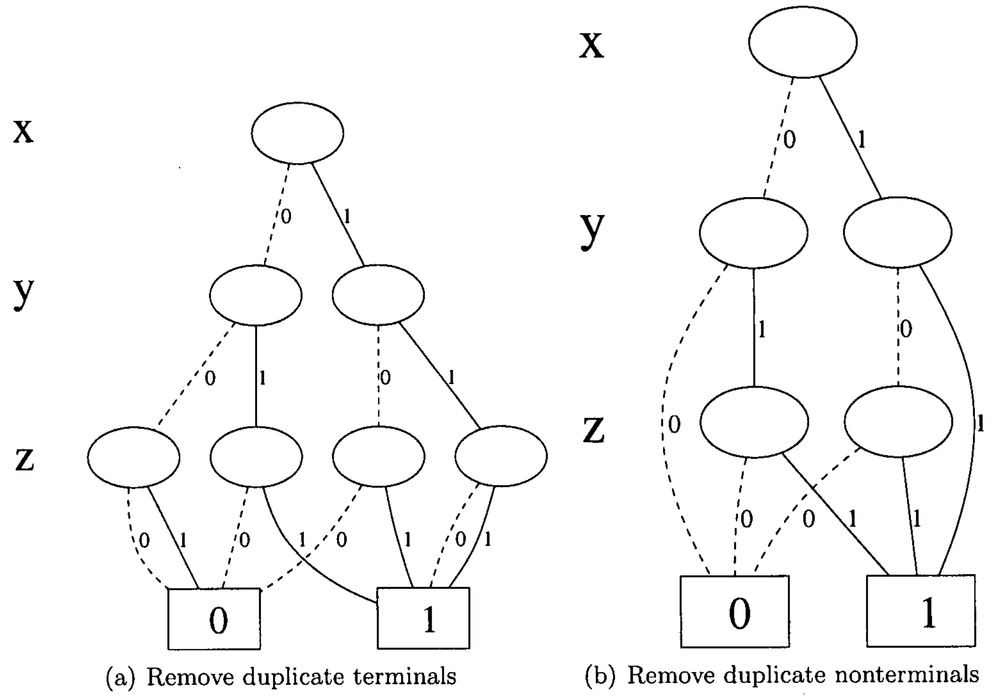


Figure 2.6: Ordered Binary Decision Tree Example

A BDD is an ordered decision tree where we remove duplicate terminals, remove duplicate nonterminals, and remove redundant vertices. i.e. all isomorphic subtrees are combined, all nodes with isomorphic children are eliminated. Figure 2.7 shows how to do these steps. After these simplifications, the BDD is canonical under a given variable ordering. The BDD for function f is given in Figure 2.7(c). The size of a BDD can be further reduced by introducing complement arcs, which point to the negation of the original function. To keep the canonicity, a complement arc can only be assigned to the pre-specified arc, i.e., we only do negation for the case when a variable is assigned to the value 0. For example, a more simplified BDD (Figure 2.7(d)) is given by introducing complement arcs to 0-arc (dotted lines).

BDDs have proven to be a successful representation for model checking on many practical systems. However, for larger industry-size systems, the BDD-based verification tools can not produce results due to the exponentially increasing BDD size, which blows up beyond the memory capabilities of most machines. This is commonly known as the *BDD blow-up problem*.


(a) Remove duplicate terminals (b) Remove duplicate nonterminals

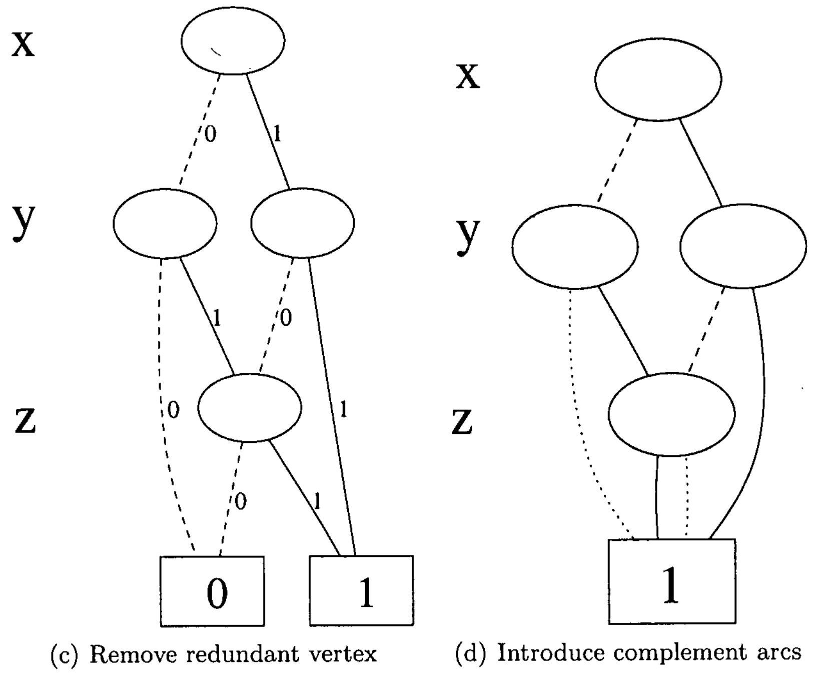
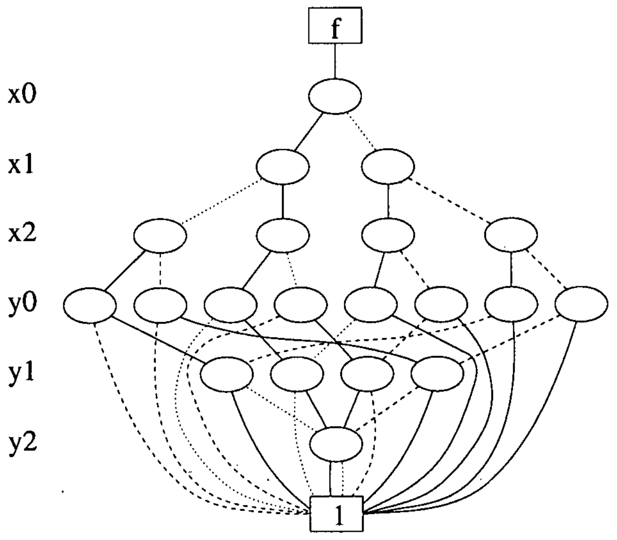
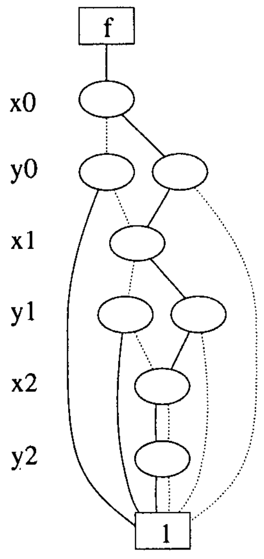

(c) Remove redundant vertex (d) Introduce complement arcs

Figure 2.7: Reduction of Figure 2.6 to BDD



(a) BDD Graph Using Ordering 0



(b) BDD Graph Using Ordering 1

Figure 2.8: BDDs with Different Orderings for Function f

Another limitation of BDD-based approaches is that the size of the BDD depends heavily on the variable ordering. For example, the 3-bit comparator function $f(x_0, x_1, x_2, y_0, y_1, y_2) = (x_0 \leftrightarrow y_0) \wedge (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$ has different BDD graphs given different variable orderings. For example, given a variable ordering $x_0 < x_1 < x_2 < y_0 < y_1 < y_2$ (Ordering 0), we have a BDD graph Figure 2.8(a); given another variable ordering $x_0 < y_0 < x_1 < y_1 < x_2 < y_2$ (Ordering 1), we have a BDD graph Figure 2.8(b). These two BDD graphs were generated using the *CUDD* package [89] and *dot* [41].

Finding the best variable ordering is an NP-complete problem. Moreover, there exist some functions (such as the middle output of a combinational circuit to multiply integers) that can not be represented efficiently regardless of the variable ordering. Several BDD ordering techniques including static and dynamic ones have been proposed to try to solve the BDD blow-up problem, e.g., [69, 79, 80].

SAT

For large systems, where BDDs blow up, Boolean satisfiability (SAT) solvers can be an alternative to manipulate Boolean formulas. SAT has received much attention by the scientific community since any NP problem can be translated into an equivalent SAT problem in polynomial time (Cook's theorem [27]). Aside from its important position in complexity theory, SAT techniques are widely used in electronic design automation, especially in the formal verification domain, e.g., such applications include ATPG (Automatic Test Pattern Generation) [20], Symbolic Model Checking [10], Bounded Model Checking [88], etc.

Unlike BDDs, the direct representation of the model as Boolean formulas does not suffer from the space explosion, but it is not canonical and requires additional efforts to check the equivalence of the formulas. SAT is still an active research area and for certain classes of problems, modern state-of-the-art SAT solvers can greatly outperform BDDs.

In the following standard definitions, e.g, [52], I will give the basic concepts of the SAT problem.

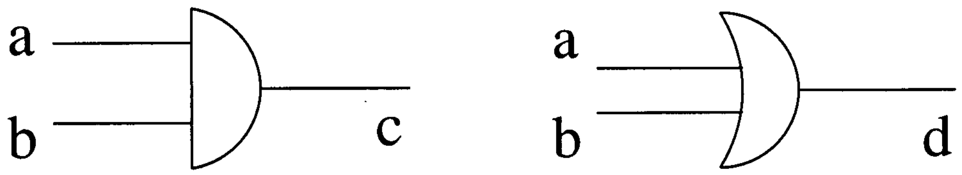
Definition 2.10 (Literal) *A literal is either a variable p or its negation $\neg p$. The first case is called a positive literal; the second is called a negative literal.*

Definition 2.11 (Clause) *A clause is a finite disjunction of literals, e.g., $l_1 \vee l_2 \vee l_3 ...$, where l_i is a literal.*

Definition 2.12 (Conjunctive Normal Form) *A propositional formula is in Conjunctive Normal Form (CNF) if it is a finite conjunction of clauses, e.g., $C_1 \wedge C_2 \wedge C_3 ...$, where C_i is a clause.*

Definition 2.13 (CNF SAT Problem) *Given a propositional formula F in CNF, the SAT problem consists of assigning values to a set of Boolean variables of F, such* *that they satisfy F, i.e., a CNF formula is satisfiable if at least one set of assignments to the variables of the formula makes it evaluate to true.*

Given the above definitions. I will use simple AND and OR gates to show how to use SAT to represent states and do equivalence checking.



(a) Two-input AND Gate (b) Two-input OR Gate

Figure 2.9: Simple Gate Examples for SAT

In Figure 2.9(a), we have a two-input AND gate. A logic formula for this gate is $(\bar{a} \rightarrow \bar{c}) \wedge (\bar{b} \rightarrow \bar{c}) \wedge (a \wedge b \rightarrow c)$ (If any input is equal to 0, the output is 0. If both inputs are equal to 1, the output is 1). After simplifying this expression, we get a CNF formula:

$$F = (a \vee \bar{c}) \wedge (b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c).$$

This propositional formula has three variables (a, b, c) and three clauses. Assignments (0,1,0), (1,0,0), (1,1,1), (0,0,0) to the 3-tuple (a, b, c) are satisfying assignments of F. These assignments essentially are the truth table for this AND gate, i.e., the state space of it.

Using the same approach, the logic formula for the two-input OR gate in Figure 2.9(b) is:

$$G = (\bar{a} \vee d) \wedge (\bar{b} \vee d) \wedge (a \vee b \vee \bar{d}).$$

Having these two propositional formulas in CNF form for our AND gate and OR gate, we can do equivalence checking of the outputs, i.e., whether c is equivalent to d given the same inputs a and b (a trivial example, but it illustrates the point). What we need to do is to check:

$$\begin{aligned}
&AND(a,b) = OR(a,b)\\
&\Longleftrightarrow (c = AND(a,b)) \wedge (d = OR(a,b)) \wedge (c = d)\\
&\Longleftrightarrow F \wedge G \wedge (d \vee \bar{c}) \wedge (\bar{d} \vee c)
\end{aligned}$$

Obviously, we need to check all possible values of a and b, i.e.,

$$\forall a, b; AND(a,b) = OR(a,b).$$

However, SAT solvers usually give a satisfying assignment for a propositional formula in CNF and can not prove the above formula directly. Fortunately, such a universal-quantified logic formula is logically equivalent to the negation of an existential-quantified logic formula. So, in practice, we check its equivalent form, i.e.,

$$\begin{aligned}
&\forall a, b; AND(a,b) = OR(a,b)\\
&\Longleftrightarrow \neg(\neg(\forall a, b; AND(a,b) = OR(a,b))\\
&\Longleftrightarrow \neg(\exists a, b; AND(a,b) \neq OR(a,b))\\
&\Longleftrightarrow \neg(\exists a, b; (c = AND(a,b)) \wedge (d = OR(a,b)) \wedge (c \neq d))\\
&\Longleftrightarrow \neg(\exists a, b; F \wedge G \wedge (d \vee c) \wedge (\bar{d} \vee \bar{c}))
\end{aligned}$$

Therefore, if we can find a satisfying assignment for

$$F \wedge G \wedge (d \vee c) \wedge (\bar{d} \vee \bar{c}),$$

it will be a counterexample to falsify the property $(c = d)$ that we want to prove. If such an assignment doesn't exist, we prove the equivalence. In our example, assignments (0, 1, 0, 1) and (1, 0, 0, 1) to the 4-tuple (a, b, c, d) are satisfying assignments for the above CNF. They are the counterexamples for the equivalence checking of our AND gate and OR gate.

The above simple example shows the basic approach of formal verification using SAT. It can be easily applied to large circuits. A combinational circuit can be represented by a conjunction of CNF formulas of its gates. A sequential circuit can be unrolled a finite number of times. Then, the resulting combinational circuit is converted to CNF and handed to a SAT decision procedure to find a counterexample whose length is less than the number of unrollings. (Note that this approach is bounded model checking (BMC), described later)

In order to find a satisfying assignment to a CNF formula, SAT procedures need to explore the search tree. In such a search tree, nodes are variables and edges are assignments (either 1 or 0 for each variable). A solution is given by a sequence of nodes and edges (variables and assignments). The search tree can be very large. Numerous techniques have been proposed to prune the search tree. Just as in BDD-based techniques, SAT solvers are also heavily effected by the variable ordering.

There is much research carried out for high-speed SAT solvers (e.g., [71, 66, 99]). This field is active and researchers are continuing to make progress toward their goals.

2.3.2 Example for Symbolic Model Checking

With the help of BDDs or SAT for Boolean formulas, we can proceed to symbolic model checking [68]. Symbolic model checking algorithms operate on sets of states instead of individual states. Therefore, some regularities in the structures of the sets can be exploited. Intuitively, the "complexity" of representing the state space is much less with symbolic representations as opposed to enumerating individual states.

In this section, I will give an example to show how to do symbolic model checking. As mentioned in the previous section, BDDs and SAT are commonly used techniques to handle Boolean formulas. Especially BDDs traditionally have been used as the underlying representation for symbolic model checkers (e.g., SMV [21], VIS [13], Bebop [5], Ever [50], etc.) for their efficiency. In this section, I will focus on BDD-based unbounded symbolic model checking. SAT-based bounded model checking will be introduced in the next section.

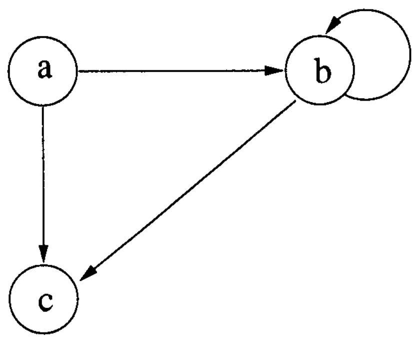


Figure 2.10: Symbolic Model Checking Example

Consider a simple example that illustrates the basic ideas of BDD-based symbolic model checking. In Figure 2.10, instead of enumerating the explicit states a, b, and c, two state variables, x_1, x_2, are introduced. We encode states as follows:

$$\begin{aligned} a &= \bar{x_1}\bar{x_2} \\ b &= \bar{x_1}x_2 \\ c &= x_1x_2 \end{aligned}$$

Therefore, the state transition relation of this state graph is

$$\begin{aligned} R &= (\bar{x_1}\bar{x_2} \wedge \bar{x_1}'x_2') \\ &\vee (\bar{x_1}\bar{x_2} \wedge x_1'x_2') \\ &\vee (\bar{x_1}x_2 \wedge \bar{x_1}'x_2') \\ &\vee (\bar{x_1}x_2 \wedge x_1'x_2') \end{aligned}$$

where x_1, x_2 represent the current states and x_1', x_2' represent the next states.

Now, we have Boolean encodings for states and the state transition relation which all can be represented as BDDs. Then, we will apply the core operation of symbolic model checking — image computation. Image computation computes the set of states that are reachable in one step from another set of states. The definitions of post-image and pre-image are as follows:

post-image$(S) := \{s' \mid \exists s.R(s, s') \wedge s \in S\}$

$$\textit{pre-image}(S) := \{s \mid \exists s'.R(s,s') \wedge s' \in S\}$$

where R is the transition relation, s, s' are states.

From the definitions, we can find that *post-image* and *pre-image* computes the successors and predecessors respectively for a set of states. In addition, from the definition of the CTL operator **EX**, we also find that **EX** is the pre-image computation.

I take the post-image computation as an example. In Figure 2.10, if we start from state a (i.e., $\bar{x_1}\bar{x_2}$), and compute the next states of a by applying the transition relation R. We just compute the conjunction $(\bar{x_1}\bar{x_2}) \wedge R$ and get $(\bar{x_1}\bar{x_2} \wedge \bar{x_1}'x_2') \vee (\bar{x_1}\bar{x_2} \wedge x_1'x_2')$. After that, we need to do *Existential Quantification* on the current variables. The definition of existential quantification is: $\exists x.f = f_{x \leftarrow 0} \vee f_{x \leftarrow 1}$. i.e., compute the cases when x is equal to 0 and 1 respectively, then combine the two cases together. In this way, we can remove the variable x. After quantifying out all the current state variables, we get the successors of state a, which satisfy $\bar{x_1}'x_2' \vee x_1'x_2'$. (In Figure 2.10, they are states b and c.) If we need to change the next states as our new current states, we also need to swap x and x' and then get $x_1x_2 \vee \bar{x_1}x_2$.

The pre-image computation is similar to the post-image computation, except quantification is on the next-state variables. Instead of computing the next states, pre-image computes previous states.

As shown in our example, post-image and pre-image computations can be used for forward reachability analysis and backward reachability analysis respectively. Reachability analysis iteratively performs the above image computations until a fixpoint is reached. In addition to its essential computation in model checking algorithms, pre-image (i.e., **EX**) can also be used in counterexample generation. Once a bad state is forward reached, the formal verification tools can give a sequence of states that leads to this bad state using backward image computation.

Image computation is one of the major bottlenecks in symbolic formal verification core. There are many techniques to address it, such as partitioned transition relation [17] and early quantification [48]. Partitioned transition relation is a technique, in which instead of a monolithic transition relation R, some small pieces of relations are given. Image computation uses these pieces to compute images, then conjoins the results. Early quantification means that instead of quantifying variables after conjoining the entire transition relation, we quantify them early when they can be safely quantified.

2.4 Industrially Scalable Verification Techniques

Symbolic model checking has been proven to be very successful when applied to hardware system [18, 19, 68]. However, the state explosion of model checking still prevents it scaling to large designs (either out of memory due to BDD blow-up or time out for SAT solvers). In industry, there are some other techniques which are widely used to attack state explosion. In the next section, I will present some of them that are related to my thesis.

2.4.1 Bounded Model Checking

We have known that in symbolic model checking, if the number of state variables is big, the BDDs for the transition relation and for sets of states are likely to be big. Symbolic model checking on such a model may become impossible due to the BDD blow-up. To avoid this problem, we can do bounded model checking instead of full model checking. Bounded Model Checking (BMC) was first proposed by Biere et al. in 1999 [10]. By using SAT procedures, it presents another approach for model checking. The basic idea of BMC is to search for a counterexample in executions whose length is bounded by some integer k. If no bug is found then BMC increases k until either a bug is found, or some pre-specified upper bound (Completeness Threshold) is reached (or in practice, until the tool runs out of time). The BMC problem can be efficiently reduced to a satisfiability problem, and can therefore be solved by SAT methods rather than BDDs.

Copty et al. [28] give a detailed comparison of performing BMC on top of a SAT solver (SIMO), and performing symbolic model checking on top of a BDD package, in an industrial setting. The designs are taken from Intel's *Pentium* 4, with over 1000 model variables. The experiments clearly show that BMC has advantages in both capacity and productivity over BDD-based symbolic model checkers for most cases. The improved productivity comes from the fact that normally BDD-based techniques need more manual guidance in order to optimize their performance. Now, BMC is a complementary technique to symbolic model checking.

However, the bound introduced in BMC also brings drawbacks for this approach. Determining whether a completeness threshold is large enough to find the inconsistencies is a hard problem. If the completeness threshold is not big enough, BMC is incomplete. And from the experiment results, if the bound k is big, BMC cannot outperform BDD-based techniques [28].

2.4.2 Symbolic Simulation

For large circuits where formal verification fails, there is simulation, which is the traditional method for testing and debugging hardware designs. In conventional simulation, one simulation run can only verify one test case. In order to fully verify a design, the simulation tool must exhaustively simulate the entire set of test cases, which is too expensive. In symbolic simulation [15], the set of test values is encoded symbolically to represent any value instead of being a specific element of the set. This allows the simulation tool to compute information on the entire set of values in a single simulation run. Such symbolic simulators can greatly accelerate the simulation.

For example, a 2-input AND gate has two inputs A and B. The output is C. In order to completely simulate the gate, a conventional simulator would try all possible input cases $\{(A = 0, B = 0), (A = 0, B = 1), (A = 1, B = 0), (A = 1, B =$

1)}. The symbolic simulator, on the other hand, treats the inputs symbolically (A, B) and gives an output $C = A \wedge B$ by one simulation run.

In order to prove the equivalence of two circuits, decision procedures need to verify the symbolic-simulation outputs, which usually are represented as Boolean expressions. Traditionally, BDDs are the most popular technique for decision procedures due to their canonicity. To avoid BDD blow-up, SAT-based techniques are another candidate for some applications. More general-purpose decision procedures, which handle logics beyond Boolean logic are also available. For example, SVC (Stanford Validity Checker) [7] is an automatic decision procedure for quantifier-free first order formulas with equality, uninterpreted functions, and arrays.

In addition to hardware verification, software verification also can be addressed by symbolic simulation. In my thesis, I need to verify software specifications vs. hardware implementations. It is possible to do symbolic simulation for both the software specifications and hardware implementations. Just as for hardware, symbolic simulation on software could greatly accelerate the simulation.

However, symbolic simulation for software introduces many challenges. The inherent exponential complexity of verification can not be avoided. Such a complexity may appear in the length of output expressions and causes memory blow up; or may appear in decision procedures and causes time out or memory blow up. Furthermore, loop handling may be impossible. If the symbolic simulator cannot determine the termination condition for a loop, it will iterate the loop infinitely which eventually causes memory blow up if there is no other technique to exit the loop.

In spite of the above limitations, symbolic simulation is an efficient approach to do verification for some problems. It is one of the techniques that I rely on for my research.

2.4.3 Combinational Equivalence Checking

Most work in my thesis leverages the success of RTL vs. gate-level equivalence checking of combinational circuits (e.g., [55] is a good survey). In this section, I will introduce some basic background for combinational equivalence checking.

The goal of combinational equivalence checking is to check whether two combinational circuits are functionally equivalent (i.e., for all possible inputs, both combinational circuits have the same outputs). The basic mathematical tool for reasoning about digital circuits is Boolean algebra (using BDD or SAT). Therefore, the equivalence of two circuits corresponds to the problem of determining the equivalence of Boolean formulas.

Combinational equivalence checking is a co-NP complete problem. Hence, finding a method with polynomial worst-case time complexity to solve this problem is extremely unlikely. However, practical instances of the problem are often more tractable. Usually, two combinational circuits being verified have a certain degree of similarity due to the steps of synthesis or optimization. There is some successful research to develop techniques that yield acceptable performance for verifying practical combinational circuits of realistic size [39, 56].

BDD-based techniques are commonly used for combinational equivalence checking. As I mentioned in previous sections, BDDs are canonical representations of Boolean formulas. By building BDDs for the two circuits, equivalence can be proven if the BDDs for the outputs of both circuits are identical.

To be specific, the BDD approach is normally as follows: For circuit C_1 and C_2, we build a circuit C which merges the primary inputs of C_1 and C_2 together and has separate primary outputs for C_1 and C_2. Then, we construct BDDs for the primary outputs of C, and compare BDDs for the corresponding primary outputs. The equivalence (or nonequivalence) of circuits is proven by the results of the comparison.

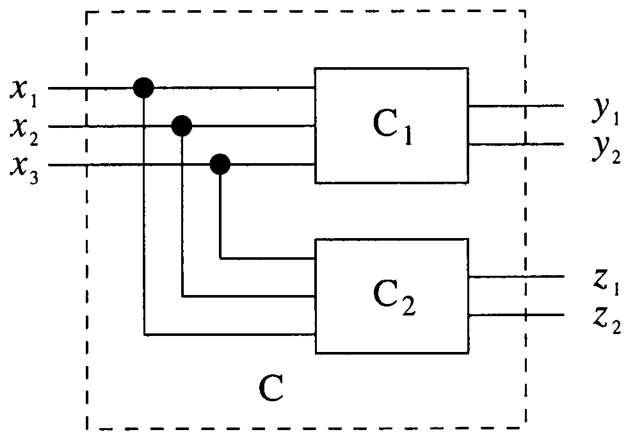


Figure 2.11: Combinational Equivalence Checking Example

For example, as shown in Figure 2.11, circuit C merges the primary inputs of C_1 and C_2 together as x_1, x_2, and x_3. The output variables are y_1, y_2, z_1, and z_2. After that, we construct BDDs for circuit C. If the BDDs for y_i and z_i are identical ($i = 1, 2$), we can declare that circuits C_1 and C_2 are equivalent.

The size of the BDDs is determined by the types of functions and the selected variable ordering. In practice, the BDDs for big circuits are beyond the memory capability of current computers. Many abstraction techniques have been applied to attack this problem. As I mentioned at the beginning of this section, the design procedure often results in some similarities of the two circuits being verified. By identifying and exploiting these structural similarities, a combinational equivalence checking tool can greatly reduce the complexity of verification.

One of these techniques is cutpoint theory [9, 12]. I use an example to describe the theory. In Figure 2.12, we have two circuits F and G. The points z_1 and z_2 are internal cutpoints for circuit F and circuit G respectively. The primary outputs of circuit F and circuit G are another pair of cutpoints called external cutpoints. If we can prove that $\forall x,\ f_1(x) = g_1(x)$, i.e., $z_1 = z_2$, we introduce another primary input variable z to substitute z_1 and z_2. Then we have $\forall z, y, f_2(z, y) = g_2(z, y) \Rightarrow F = G$. i.e., if we can prove all the cutpoints are equivalent, so are the functions. In general, the method is conservative (i.e., success proves equivalence, but failure doesn't prove inequivalence) because constraints on the cutpoint "inputs" are lost. The situation that two circuits are equivalent but verification reports inequivalence is called *false negative* or *false inequivalence.* Minimizing this false inequivalence/negative problem has been an active research area. The general solution is to re-introduce constraints on the cutpoints, either in advance [12] or as needed [56, 62].

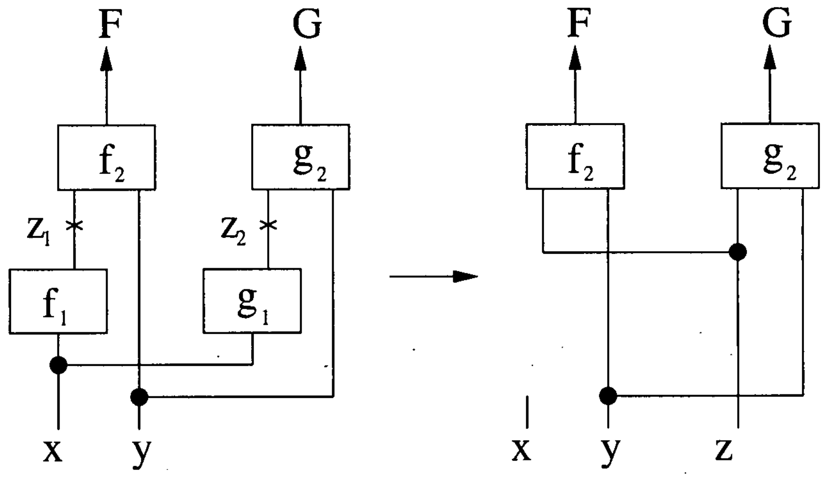


Figure 2.12: Cutpoint Example

2.4.4 GSTE

As mentioned in the section on symbolic simulation, there are many symbolic simulation techniques available. Among them, is GSTE (*Generalized Symbolic Trajectory Evaluation* [96, 97, 95]), developed at Intel. GSTE has been used extensively and successfully [8]. The user can write specifications in a restricted temporal logic specifying the behavior over trajectories (sequences of circuit states), then verifies the implementation with respect to the specifications. The specification in GSTE is described as an automaton, to be specific, an *assertion graph.* The goal of GSTE verification is to prove that a hardware implementation obeys an assertion graph.

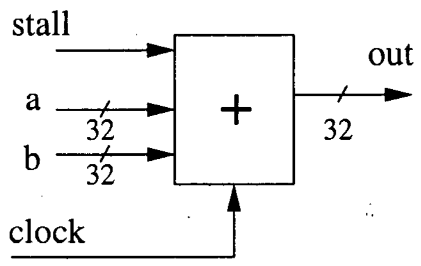


Figure 2.13: Generic Example Circuit

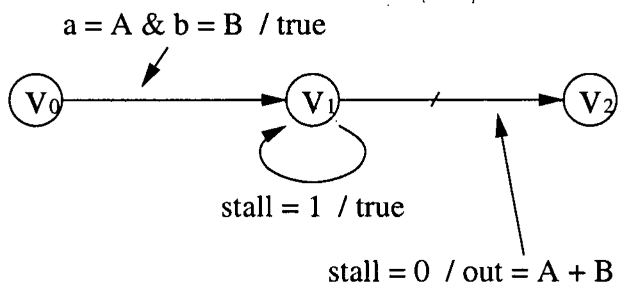


Figure 2.14: Generic Example Assertion Graph

The assertion graph is a variant of ∀-automata [65]. By introducing assertion graphs, GSTE significantly extends classical symbolic trajectory evaluation [85]. Such an extension gives GSTE more expressive power. At Intel, GSTE has proven to be a success on leading-edge designs (e.g., [8, 83]). Details on the theory of GSTE and assertion graphs can be found elsewhere [96]. In this section, I will give a simple assertion graph example to show the underlying ideas of GSTE.

Figure 2.13 gives a simple sequential pipelined adder with two data inputs a and b, a stall input *stall*, and a data output *out*. This adder loads input data at the first cycle, and outputs the result $a + b$ at the second cycle. If the stall signal is enabled, the adder will hold the result until *stall* is disabled.

Figure 2.14 gives an assertion graph for this pipelined adder with each edge corresponding to a clock cycle. Each edge in the assertion graph is labeled with an *antecedent* and a *consequent*, which are Boolean formulas of the circuit signals. e.g., the label for $edge(v_1, v_2)$ is $stall = 0/out = A + B$. The part before "/" is the antecedent and $out = A + B$ is the consequent. In this example, A and B are 32-bit input data. Furthermore, some edges can be labeled as *terminal edges* ($edge(v_1, v_2)$ is a terminal edge in my example). Every path starting from the initial vertex (vertex v_0 in this example) and ending on a terminal edge represents a distinct temporal assertion. For example, the path that goes from v_0 to v_1, loops back to v_1, and then proceeds to v_2 corresponds to the temporal assertion "**If** $a = A \& b = B$ holds on the first cycle, **and** $stall = 1$ holds on the second cycle, **and** $stall = 0$ holds on the third cycle. **then** $out = A + B$ must hold on the third cycle." If at least one antecedent fails (i.e., the assertion is satisfied vacuously due to a failure of the preconditions) or all antecedents and consequents are satisfied on a path of the assertion graph, we can say that the run of the circuit satisfies this path. If a circuit run satisfies every path from initial vertex to terminal edge, it satisfies the assertion graph.

2.5 More Abstraction Techniques

In this section, I will present additional abstraction techniques that are used in related work to address the equivalence checking of software specification vs. hardware implementations.

The basic idea of such abstractions is: if the property holds on the abstract model, it also holds on the concrete model. If the property doesn't hold on the abstract model, the model checker uses the counterexample to re-simulate on the concrete model. If the simulation is successful, a concrete counterexample is found, otherwise, the abstract counterexample is spurious, and the abstraction needs to be refined. This procedure is called Counterexample Guided Abstraction Refinement (CEGAR) [4, 25, 6].

2.5.1 Uninterpreted Functions

Usually, symbolic simulation tools can use uninterpreted functions to improve efficiency. Uninterpreted functions are a powerful abstraction mechanism for symbolic simulation. In this approach, meaningless function symbols replace actual computations. The function symbol is "uninterpreted" and can represent any function, similar to how a variable symbol represents any value. Therefore, verification can avoid the complex details of the actual computations.

For example, if $a = x$, $b = y$, and f is an uninterpreted function, then we know that $f(a, b) = f(x, y)$ without knowing the meaning of the function f.

Decision procedures (e.g., SVC [7], CVC [90] UCLID [16]) based on uninterpreted functions are available and have proven successful for many verification problems.

2.5.2 Predicate Abstraction

For the software specifications against hardware implementations verification problem, if the two models are too big for verification tools to handle, we need to create abstract models that are amenable to available model checking techniques. Predicate abstraction [43] is a technique that arises from the software verification domain. For example, Ball and Rajamani at Microsoft [4, 6] employ this technique in the SLAM project to verify that Windows device drivers obey API conventions. The SLAM project found many bugs in large software program. Inspired by the great success of such a method in software, Jain, Kroening, and Clarke [54] have introduced predicate abstraction into the software specifications against hardware implementations verification problem. Predicate abstraction shows promise to reduce the complexity of the models.

The approach of predicate abstraction is to use Boolean variables to replace the variables of the concrete model. These Boolean variables correspond to predicates of the concrete model.

For example, here is a simple piece of C code. It is a concrete model. The "..." parts represent code that doesn't change the values of variables p and q, and doesn't effect the control flow of the program.

```
function(){
  int p, q;
  ...
  p = 1;
  q = rand()*10 - 5;
  ...
  while (p >= 0){
    ...
    if (q < 0)
      p = p - q;
    else
      p = p + q;
    ...
  }
  ...
}
```

We want to prove that the program can not terminate. The set of predicates can be $\{p \geq 0, q < 0, q \geq 0\}$. Associated with each predicate in the above set, we have Boolean variables a_1, a_2, and a_3. e.g., a_1 is *true* if and only if $p \geq 0$ is *true.*

Therefore, the original program can be abstracted into a Boolean program that only has these above Boolean variables.

```
function(){
  bool a1, a2, a3;
  a1 = true;
  a2 = rand{true, false};// a2 can be randomly initialized
                           // to true or false.
  while (a1){
    if (a2)
      a1 =  true;
    if (a3)
      a1 =  true;
  }
}
```

Using the three Boolean variables as the state variables, we can have a state-transition graph for this abstract model. The verification problem is abstracted to a reachability problem — is there a path from the initial state $(a_1 = true, a_2 = X, a_3 = X)$ to $(a_1 = false, a_2 = X, a_3 = X)$?[4]

In our example, the forward reachability analysis on the abstracted model can prove $a_1 = true$ for all the reachable states, so non-termination of this program is proven.

We can see that predicate abstraction can greatly reduce the state space and enable property verification for big systems. However, such an abstraction is very coarse — it abstracts software to Boolean programs, which restricts the properties that can be checked. In addition, finding the set of predicates to prove a property is a highly challenging job. In some cases, domain experts need to be involved to provide good heuristics.

[4]X denotes a "don't care" — it can be either *true* or *false*.

2.6 Program Analysis

This section provides the second part of the background for the thesis, program analysis.

Since I want to prove the equivalence of software specifications and hardware implementations, I need to study not only hardware but also software. The natural way that a software specification expresses a functionality is very different from the way hardware does. For example, given a multiplication function that is executed in one cycle, the software specification will naturally give a loop with shift-add computations, and the hardware implementation, on the other hand, will parallelize all the additions of partial products. Although there are many differences, fundamentally, the software specification and the hardware implementation are both just computations of a functionality — either in serial or in parallel. In order to understand how serial computations in software can be executed on hardware in parallel, we need to analyze software specifications and identify the instructions which can be reordered or executed in parallel, i.e., we need program analyses which can give execution-order constraints for instructions. Usually, such constraints are approximations of the precise constraints. This is because program analysis is a static technique which analyzes software without actually running it.

In this section, I will give some basic definitions of program analysis — dependence analysis (to be specific, control dependence and data dependence analysis) and data flow analysis. These are standard definitions; more details and advanced topics can be found in [59, 72].

I would like to give the definition of control flow graph first.

Control Flow Graphs

A control flow graph (CFG) is a graph G = (V,E), with V the set of vertices, and E the set of edges. Each vertex is a block of straight-line code (a basic block), and an edge is a jump in the control flow. Multiple outgoing edges from a vertex indicates conditional branching. CFGs are a standard way to describe the behavior of software, representing all alternatives of control flow.

With CFGs to represent the software, I continue to give dependence analysis and data flow analysis.

2.6.1 Dependence Analysis

For dependence analysis, I will cover control dependence and data dependence.

Control Dependence

Control dependence identifies the conditions that may affect instructions, which is very useful for instruction scheduling. In addition, such control dependences may appear in hardware implementations as control logic, so they could be possible correspondences between software and hardware.

Definition 2.14 (Control Dependence) *An instruction i_2 is control dependent on another instruction i_1 if and only if i_2 is conditionally guarded by i_1.*

For example, in the following code,

```
if (x > 0) foo();
else bar();
```

the call of the foo function is control dependent on the condition $x > 0$, The bar function is control dependent on $x \leq 0$. In a hardware implementation, the condition might be an internal wire c (for logic $x > 0$), the logic for the above functionality might be $(c \wedge foo()) \vee (\bar{c} \wedge bar())$.

Data Dependence

Other than control dependence, we need data dependence to show relations when two instructions read or write the same variable. According to positions of the two instructions and whether these instructions write or read the variable, there are four classes of data dependence:

- flow dependence (read after write)
- antidependence (write after read)
- output dependence (write after write)
- input dependence (read after read)

Definition 2.15 (Flow Dependence) *An instruction i_2 is flow dependent on i_1 if and only if i_1 writes a variable that i_2 reads and i_1 precedes i_2 in execution.*

The following is an example of a flow dependence:

```
i1        x = 0;
i2        y = x + 1;
```

Definition 2.16 (Antidependence) *An instruction i_2 is antidependent on i_1 if and only if i_2 writes a variable that i_1 reads and i_1 precedes i_2 in execution.*

The following is an example of an antidependence:

```
i1        x := y + 1;
i2        y := 0;
```

Here, i_2 writes the value of y but i_1 reads the old value of y.

Definition 2.17 (Output Dependence) *An instruction i_2 is output dependent on i_1 if and only if i_1 and i_2 write the same variable and i_1 precedes i_2 in execution.*

The following is an example of an output dependence:

```
i1        x := 0;
i2        x := 1;
```

Here, i_2 and i_1 both write the same variable x.

Definition 2.18 (Input Dependence) *An instruction i_2 is input dependent on i_1 if and only if i_1 and i_2 read the same variable and i_1 precedes i_2 in execution.*

This is not a dependence compared to the other dependence relations, because it does not constrain the execution order of two instructions.

The following is an example of an input dependence:

```
i1        y := x + 1;
i2        z := x + 2;
```

In this example, i_2 reads the variable x after i_1 reads it. Thus, i_2 and i_1 don't depend on each other. Some compiler optimizations can make use of this input dependence to reorder instructions. In hardware implementation, such computations can be scheduled in parallel.

2.6.2 Data Flow Analysis

In addition to dependence analysis, we also need data flow analysis to track how software manipulates its data, which we can use to simplify the software and bridge the gap between software and hardware. For example, we can do dead-code elimination, constant propagation, and path merging. In this section, I will explain forward analysis and backward analysis, and use reaching definitions and live variables analysis as examples respectively.

Forward Analysis

In forward analysis, data flow analysis determines certain properties of a program in the direction of program execution (i.e., forward). Usually, such a data flow analysis will compute a fixpoint of basic blocks on the CFG by using information passing from the predecessors of a block to the block itself.

Example: Reaching Definitions

Definition 2.19 (Reaching Definitions) *A "definition" of a variable is an instruction that either initializes or assigns the variable. Reaching definitions give the set of possible definitions which can reach a given point in the program.*

From the above definition, we can see that a definition d reaches a point p if there exists a path from the point d to the point p and the definition d is not canceled (or killed) by some other definitions along that path.

The algorithm that computes reaching definitions uses forward analysis and conservatively simulates the program until it reaches a least fixed point. Before giving the algorithm, I define:

B is a basic block on the control flow graph for the program;

GEN(B) is the set of definitions that are generated at B, i.e., it is the set of definitions within B, and such definitions are not subsequently followed by other definitions to the same variables within B;

KILL(B) is the set of definitions that are killed at B, i.e., for a definition in KILL(B), there is a definition to the same variable within B.

With such definitions, the algorithm for reaching definitions is shown in Figure 2.15. We can see that at each iteration, the computation is from the predecessors of current block to itself, i.e., the direction of computations is forward.

```
 1: function ReachingDefinitions($CFG$) /* Compute reaching definitions at the
    entrance for each block ($B$) of $CFG$ */
    /* Initialize $RD$ and $taskQueue$ */
 2: $taskQueue := \emptyset$;
 3: for all blocks $B$ of $CFG$ do
 4:   $RD(B) := \emptyset$;
 5:   add $B$ into $taskQueue$;
 6: end for
    /* update $RD$ and $taskQueue$ */
 7: while $taskQueue \neq \emptyset$ do
 8:   $taskQueue1 := \emptyset$;
 9:   for each block $B$ of taskQueue do
10:     remove B;
        /* Pred(B) is the set of predecessors of B */
11:     $RD(B) := \bigvee_{B' \in Pred(B)}(GEN(B') \vee (RD(B') - KILL(B')))$;
12:     if there is a change in $RD(B)$ then
13:       put all successors of $B$ into $taskQueue1$;
14:     end if
15:   end for
16:   $taskQueue := taskQueue1$;
17: end while
```

Figure 2.15: Algorithm for Reaching Definitions

Here is a simple example.

```
1: cout = 2;
2: if (foo)
3:     bar = 1;
4: else
5:     bar = 2;
6: endif
7: while (cout <= 4)
8:     cout +=bar;
```

The reaching definitions at line 6 are the instructions $\{i_1, i_3, i_5\}$. The reaching definitions at line 7 are the instructions $\{i_1, i_3, i_5, i_8\}$

We can use the result of reaching definitions to partition the state space to minimize communications between sub-state spaces (e.g., [1]). For example, in the above example, at line 6, the value of variable "bar" is defined by its reaching definitions $\{i_3, i_5\}$. We can case-split the consequent executions into two cases: get the definition from line 3 (i.e., "bar = 1"), or from line 5 (i.e., "bar = 2").

Backward Analysis

Unlike forward analysis, which computes a fixpoint using information passing from the predecessors of a block to the block itself, backward analysis computes information from the successors of the block, i.e., the direction of computations is backward. I use live variable analysis as an example to show backward analysis.

Example: Live Variables Analysis

Definition 2.20 (Live Variables) *A variable is live at a given program location if the value of the variable is used along some path starting at this location.*

The live variable analysis determines whether a variable at a program location may be potentially read afterwards before its next write update. It is useful for my research because such live variables in software specifications might be allocated to registers in hardware implementations. In addition, I can combine information from live variables and reaching definitions to do path merging. For example, in the following code,

```
1:  wrap = 0;
2:  while (wrap <=3){
3:    cout[wrap] = 1;      /* live = {wrap, in} */
4:    if (in[wrap] == 10)
5:      wrap +=3;
6:    else
7:      wrap ++;
8:  }
9:  cout[wrap] = 1;
10: exit();
```

the set of live variables at line 3 is {wrap, in}, i.e., any other variables are not read in future computations. Therefore, any two computation paths that reach line 2 and have the same the values for both "wrap" and "in" can be safely merged. This technique is very useful to simplify software when there are an exponential number of paths in software specifications.

The algorithm to compute live variables is almost the same as Figure 2.15, except the direction of computations.

Before giving the algorithm, I re-define the meaning of GEN and KILL functions:

GEN(B) is the set of variables that are used (read) within B prior to any definition of that variable within B;

KILL(B) is the set of variables that are written within B prior to any read of the variable within B;

With such definitions, the algorithm for live variables at the exit point of a basic block is shown in Figure 2.16.

```
1: function LiveVariables($CFG$) /* Compute live variables at the exit point for
   each block ($B$) of $CFG$ */

   /* Initialize $LV$ and $taskQueue$ */
2: $taskQueue := \emptyset$;
3: for all blocks $B$ of $CFG$ do
4:   $LV(B) := \emptyset$;
5:   add $B$ into $taskQueue$;
6: end for

   /* update $LV$ and $taskQueue$ */
7: while $taskQueue \neq \emptyset$ do
8:   $taskQueue1 := \emptyset$;
9:   for each block $B$ of taskQueue do
10:    remove B;
       /* Succ(B) is the set of successors of B */
11:    $LV(B) := \bigvee_{B' \in Succ(B)}(GEN(B') \vee (LV(B') - KILL(B')))$;
12:    if there is a change in $LV(B)$ then
13:      put all predecessors of $B$ into $taskQueue1$;
14:    end if
15:  end for
16:  $taskQueue := taskQueue1$;
17: end while
```

Figure 2.16: Algorithm for Live Variables

We can see that, compared to the algorithm for reaching definitions, the algorithm for live variables swaps predecessors and successors. Therefore, it changes the direction of computations from forward analysis to backward analysis.

Chapter 3

ACFG-Based Partitioned Model Checking

This chapter is based on a paper that was published in the 10th Asia Pacific Design Automation Conference (ASPDAC'05) [38]. I propose a natural way to formalize a cycle-accurate software specification as an *annotated control flow graph* (ACFG), and then I introduce the first partitioned model checking approach that exploits the ACFG for formal equivalence checking of software specifications vs. sequential hardware implementations. Experimental results show that my new method is faster than standard model checking.

3.1 Introduction

In Section 2.4.4, I mentioned the efficiency of GSTE model checking. GSTE has proven very successful in industry [8], but GSTE is fundamentally an approach for circuit verification. It doesn't handle complicated software/hardware verification, because assertion graphs were not designed to model software. For example, assertion graphs contain "symbolic constants" that are used somewhat like local variables, but the value of a symbolic constant does not change, whereas the basic idea of software and CFGs is the assignment of new values to variables — the assertion graph is declarative, whereas software is imperative. Furthermore, there are no internal variables in an assertion graph; the circuit and the assertion graph share all the same state variables. For a specification written in software, the circuit and the specification need to have their own, separate variables. To solve these problems, I combine ideas from CFGs and GSTE assertion graphs to form my ACFG structure.

3.2 Related Work

In this section, I give the related work, which has two parts. One is the related work on formal verification of software specifications vs. sequential circuits. The other is the related work on partitioned model checking.

3.2.1 Software Specifications vs. Sequential Circuits

Researchers have previously considered formal verification of software specifications vs. hardware implementations. I will introduce the related work on combinational circuits in Chapter 5. In this section, I survey only the related work on sequential circuits.

Séméria et al. [86] reported verifying C against Verilog as part of a C-based design flow. However, their C model was already in RTL C, so the problem is basically RTL-to-RTL verification, and standard RTL-to-RTL commercial tools were used.

When the software specifications are higher-level than RTL, more complex approaches are needed. Verification tools can put restrictions on the problem, and give feasible solutions.

If the high-level and low-level models are very close, there are some techniques that exploits the similarity. For example, Matsumoto, Saito, and Fujita compare two C-based hardware descriptions [67]. In order to verify large C descriptions efficiently, they rely on scanning for textual differences to reduce problem complexity, then enumerate execution paths and apply symbolic simulation and word-level uninterpreted functions.

If the software is arbitrary, high-level code, then full formal verification is undecidable, but bounded-length verification is possible using symbolic execution [22, 23, 61]. Kroening, Clarke and Yorav [61] apply BMC (Bounded Model Checking) to both a circuit and a C program. Their tool provides full support for arbitrary code in full ANSI-C, also via path enumeration and symbolic simulation, but at a fully bit-accurate level and using SAT as the computation engine. However, this method shows only the absence of inconsistencies up to a given bound. Furthermore, the method enumerates all execution paths in the software, and the number of paths grows exponentially in the number of branches. In addition, no abstraction techniques are used to reduce the state space.

In order to avoid the state space explosion problem of full formal verification, Jain, Kroening, and Clarke [54] introduce predicate abstraction for hardware implementations against software specifications. This approach can greatly reduce the size of the state space and verify certain properties for large circuits. The strength of that work is powerful abstraction techniques that reduce the complexity of the software specifications. However, such abstraction techniques can be too coarse, and finding good predicates is highly challenging. These weaknesses restrict the properties that can be checked, and it is not suitable for many instances of equivalence checking of software specifications vs. hardware implementations. In addition, they use standard symbolic model checking for the verification. In contrast, I concentrate on improving the core model checking. I believe that both abstraction and improved model checking are necessary in practice.

3.2.2 Partitioned Model Checking

I categorize ACFG-based model checking as a partitioned model checking approach. Partitioned model checking is an approach to attack the state space explosion problem. Such an approach dynamically breaks down a large model checking data structure into several smaller data structures (partitions), then implicitly combines the partitions to represent the large model checking data structure.

For example, Burch, Clarke, and Long [17] gave an approach to represent the transition relation by using a list of BDDs instead of using a monolithic BDD. BDDs on the list are implicitly conjoined/disjoined. Many optimizations, such as early quantification [48] (see Section 2.3.2), can be done for image computations.

Instead of partitioning the transition relation, Hu, York, and Dill [49, 51] used a list of BDDs for the set of reachable states instead of a monolithic BDD. Each BDD on the list represents a partition. Therefore, the BDDs on the list are implicitly conjoined to represent the large monolithic BDD. By using implicitly conjoined BDDs, they can avoid BDD blow-up in some cases. In addition, BDD image computations (To be specific, in [49, 51], the backward image computation for the CTL operator AX) might be localized to each partition, so, image computations can be faster. Narayan, Jain, Fujita, and Sangiovanni-Vincentelli [73] also divided the reachable states into several partitions. BDDs for different partitions are allowed to have different variable orderings. Therefore, building BDDs for all partitions can use much less memory than just building a monolithic BDD.

In [84], which appeared contemporaneously with my work, Sebastiani et al. showed that GSTE is partitioned model checking (GSTE partitions reachable states, and the partitionings are driven by the property being verified). My approach takes advantage of this GSTE-style partitioning, and further exploits the control flow graph of the software specification to do partitioned model checking.

3.3 Annotated Control Flow Graph

In order to harness GSTE for software specifications, I combine ideas from CFGs and GSTE assertion graphs to form my ACFG structure. As in GSTE assertion graphs (see Section 2.4.4), the edges are labeled, rather than the vertices, as in CFGs. (CFGs are a standard way to describe the behavior of software. See Section 2.6.)

However, paths through an ACFG, like paths through a standard CFG, correspond to control flow through a program. The labels include antecedents and consequents, as in GSTE assertion graphs. However, I also introduce *assignments*, which assign new values to variables.

For example, Figure 3.1 shows a simple ACFG which describes a counter circuit that counts up to 4. The ACFG has a 1-bit input `load`, a 3-bit input `D`, and a 3-bit state variable and output `X`. (The circuit being verified is assumed to have the same I/O signals, but it's internal state might be different. The user declares which variables are inputs, outputs, and state variables, as well as the correspondence between ACFG and circuit signals.) Each edge label has three parts: antecedent/assignment/consequent. The concept of antecedents and consequents come from GSTE: they are Boolean-valued formulas over the variables. Intuitively, the antecedent is a condition that must be true in order to follow that edge, and the consequent is a condition that the verification tool must check if the computation follows that edge. The assignment computes values for the variables and updates them. The example ACFG loads a 3-bit number, saves it to the internal variables $X[0..2]$ at the entry vertex (vertex 0). $X[0..2]$ are also output variables.

After the number is loaded, the control flow of the ACFG will jump to vertex 1, and branch based on the value of the number. If the number is less than 4, it increases by one for each cycle until it hits 4. If the current counter output is equal or greater than 4, it holds that value for consequent cycles on the terminate vertex (vertex 2 on the graph).

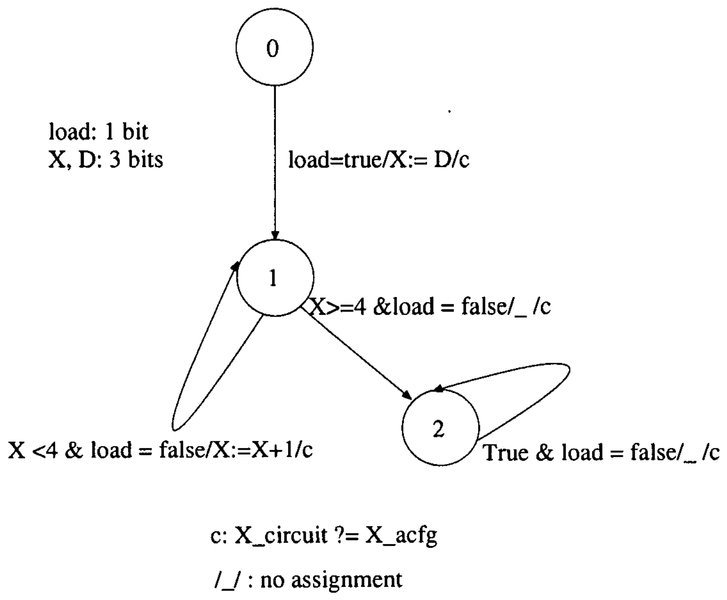


Figure 3.1: ACFG for a Counter Circuit

More formally, an ACFG is an edge-labeled, directed graph, with a distinguished initial vertex v_I. Each edge e is labeled with an antecedent $ant(e)$, an assignment $assign(e)$, and a consequent $cons(e)$. Both $ant(e)$ and $cons(e)$ are propositional formulas over inputs and state variables. The assignment $assign(e)$ associates a formula over the current variables to each state variable. (For variables that do not change value on an edge, the formula is simply the variable itself.) When an assignment is executed, the formulas are evaluated using the current values of the variables, and then the variables are updated with the value of the associated formula. When the flow of control is at a vertex v_i, an edge e $(v_i \rightarrow v_j)$ can execute the assignment $assign(e)$ iff it satisfies the ant(e). The consequent $cons(e)$ is a property that is expected to hold after the $assign(e)$ execution.

Intuitively, an ACFG is used as a reference model for the hardware. In my examples, only input variables are shared between the ACFG and the hardware. The ACFG and the hardware each have their own internal state variables and output variables. Therefore, *assignments* cannot update the state variables of the hardware and cannot update the shared input variables either. *Antecedents* can read ACFG state variables and shared input variables only.[1] However, in order to check the consistency of the reference model and the hardware, *consequents* can read all variables of the ACFG and the hardware. We can see that *antecedents* and *consequents* are read-only, while *assignments* can read and write variables.

In order to map the cycle-accurate-execution of circuits to ACFGs, I define that each edge of the ACFG takes one clock cycle. For convenience, I allow a sequence of assignment statements on each edge, but the entire sequence is executed in one clock cycle. In general, any software that has a finite number of paths can be put on an edge label as an assignment (e.g., if-then-else, loops that can be completely unrolled, etc.).

With the above definitions, I give the following acceptance condition of an ACFG. A trace of the circuit under verification obeys the property specified by the ACFG graph if and only if on every finite path (starting from the entry vertex) of the same length as the trace, if the antecedents on all edges are satisfied, the consequents are satisfied as well. Like GSTE, the ACFG is a variant of $\forall$-automata [65], i.e., the model checking algorithm check a trace of the circuit against all paths in the ACFG.

3.4 ACFG-Based Model Checking

I start by first defining a simple computation model. Let I_G, S_G, and O_G be the sets of Boolean input, state, and output variables in the ACFG. Similarly, let I_C, S_C, and O_C be those in the implementation circuit model ($I_G = I_C$). For simplicity, I assume that there is a one-to-one mapping between the interface (i.e., input, output variables) of the high level model captured by the ACFG and the interface of the circuit model. Model M is the product model of the ACFG specification and the

[1]This restriction can be relaxed if the ACFG is used to represent properties (instead of having a reference model) of the hardware. I conjecture that the ACFG-based model checking algorithm in the next section would still work for such ACFG representations.

circuit, i.e., $M = G \times C$. A state in a model M is simply a Boolean assignment to the input and state variables $V_M = I_M \cup S_M$ in the model ($I_M = I_G = I_C$, and $S_M = S_G \cup S_C$).

For the ACFG, the set of assignments *assign*(e) on each edge e defines a state transition relation in the software specification associated with e. From now on, I shall denote such a transition relation as *assign*(e)(V_G, S'_G) where S'_G is a copy of S_G for holding the values after the transition and $V_G = S_G \cup I_G$. We can see that we don't allow any assignments on input variables. For the circuit model, let $R_C(V_C, S'_C)$ be the single state transition relation where $V_C = S_C \cup I_C$.

To model check the circuit implementation against the ACFG, the standard approach is to build the single product automaton M combining the ACFG and the circuit model, and then to perform reachability analysis on the combined model. However, such an approach is prone to the state explosion problem.

My model checking solution, instead, is based on GSTE-style model checking proposed in [96]. The key idea is to partition the software states and the circuit states based on the structure of the ACFG, and use the flow and the antecedents in the ACFG to guide and tailor the state space exploration. More specifically, I would like to compute a simulation relation *sim*(e)(V_G, V_C) for each edge e in the ACFG, such that the simulation relation tells us (1) what is the set of states the software specification can be in on the particular edge e, and (2) for each such state, what is the corresponding set of possible states in the circuit model. The consequent on the edge is then checked against the simulation relation.

Before describing the model checking algorithm, I first define two partial post-image transformers (each does a part of post-image computation for model M). The first one is the post-image computation for the ACFG software specification. It computes the set of states that are reachable by one edge of the ACFG from the current state S_G. (The circuit states are unchanged, since *assign*(e) involves only variables for the ACFG software specification.) Given a global state predicate

$P(V_G, V_C)$, the partial post-image transformer of the predicate with respect to an edge e in the ACFG is:

$$\begin{aligned}&\mathit{post}(e)(P(V_G, V_C)) = \\ &\quad (\exists V_G . P(V_G, V_C) \wedge \mathit{assign}(e)(V_G, S'_G))[S'_G/S_G]\end{aligned}$$

where S'_G/S_G denotes the substitution of every variable v' in copy S'_G with its original variable v in S_G. Similarly, the partial post-image transformer of the predicate with respect to the circuit is

$$\begin{aligned}&\mathit{post}_C(P(V_G, V_C)) = \\ &\quad (\exists V_C . P(V_G, V_C) \wedge R_C(V_C, S'_C))[S'_C/S_C].\end{aligned}$$

It computes the set of states that are reachable in one cycle from the current S_C. (ACFG states are unchanged, since R_C involves only variables for circuits.) Therefore,

$$\mathit{post}(e)(\mathit{post}_C(P(V_G, V_C)))$$

is the set of states that are reachable in one cycle for the product model M.

Figure 3.2 lists my GSTE-style model-checking algorithm in its entirety. Line 4 applies antecedents from the ACFG to the circuit being verified. The rest of the lines 2–12 simply initialize the computation, with initial task queue entries from the initial vertex. The main verification loop in lines 13–24 propagates newly discovered relations between graph edges and circuit states in an event-driven manner, until a fixpoint is reached (or a bug is found). The typical $\mathit{cons}(e)$ checked in line 15 would be that the corresponding outputs agree, but more general properties can be verified. The partitioning occurs because I compute separate BDDs for each $\mathit{sim}(e)$.

Because the algorithm is very similar to GSTE-model checking, I hope in the future to exploit GSTE-style abstraction for higher capacity.

```
1: function ModelCheck($ACFG$, $post_C$)
   /* Incorporate mapped antecedents to the circuit and
      initialize the simulation relation */
2: for all edges $e$ of $ACFG$ graph do
3:   /* applies antecedents to both ACFG and the circuit */
4:   $ant(e) := ant(e) \wedge ant(e)[I_G/I_C][O_G/O_C]$;
5:   if $e$ is from the entry vertex then
6:     /* $sim(e)$ is the set of reachable states of the product model at edge $e$.*/
7:     $sim(e) := post(e)(ant(e))$;
8:     add $e$ into taskQueue;
9:   else
10:    $sim(e) := \emptyset$;
11:  end if
12: end for
   /* Compute simulation relation and check consequents */
13: while taskQueue $\neq \emptyset$ do
14:   remove an edge $e$ from taskQueue;
15:   if $sim(e) \not\Longrightarrow cons(e)$ then
16:     return(a counter-example trace);
17:   end if
18:   for each successor edge $e'$ of $e$ do
19:     $sim(e') := sim(e') \vee post(e)(post_C(sim(e)) \wedge ant(e'))$;
20:     if there is a change in $sim(e')$ then
21:       put $e'$ into taskQueue;
22:     end if
23:   end for
24: end while
25: return(succeed);
```

Figure 3.2: A Partitioned Model Checking Algorithm for ACFG

3.5 Experimental Results

I have conducted experiments to test the performance of my model-checking algorithm. As my test case, I use a radix-2 SRT unsigned fractional division circuit. (An example circuit with 8-bit dividend and 4-bit divisor in modified ISCAS'89 format can be found at Appendix B). Figure 3.3 shows the algorithm.[2] There are several reasons for choosing this circuit. One reason is that the SRT algorithm is well-known and widely used in practice, and the circuit is fairly small, but the specification still has for-loops and if-then-else branches. Modeling such kinds of control flow is the goal of using ACFGs. Another reason is that the example is scalable, so I can test the capability of my verification tool. I have a parameterized circuit generator tool that builds a divider circuit with $2N$-bit dividend and N-bit divisor for any N. As is done in practice, the circuit uses a redundant representation for the partial remainder to avoid needing a long carry chain, which is the advantage of SRT division.

By using an ACFG to describe the algorithm, I can have local variables for the specification, and a flexible control-flow graph structure to describe the control flow. I define the single-cycle operations on each edge. The operations are an ordered sequence of assignments. As noted earlier, these sequences are just a convenience to make it is easy to define a complicated, single-cycle operation without writing a hard-to-understand single-assignment. The key idea is that the specification model should be easy to maintain and change.

[2]SRT division in general allows many implementations, because the redundant quotient representation allows some freedom in selecting the next quotient digit. This thesis is about equivalence checking, so I check the equivalence between specific implementations of the software and the hardware. Thus, Figure 3.3 gives one particular implementation of the SRT division.

```
function SRT_Division($dividend, divisor, N$)
$P^0 = dividend$;
for $i = 1$ to $N$ do
  if $2P^{i-1} \geq 0.5$ then
    $q_{N-i} = 1$;
    $P^i = 2P^{i-1} - divisor$;
  else
    if $2P^{i-1} < -0.5$ then
      $q_{N-i} = -1$;
      $P^i = 2P^{i-1} + divisor$;
    else
      $q_{N-i} = 0$;
      $P^i = 2P^{i-1}$;
    end if
  end if
end for
if $P^N < 0$ then
  $P^N = P^N + divisor$;
  $q = q - 1$;
end if
```

Figure 3.3: SRT Unsigned Fractional Division Algorithm

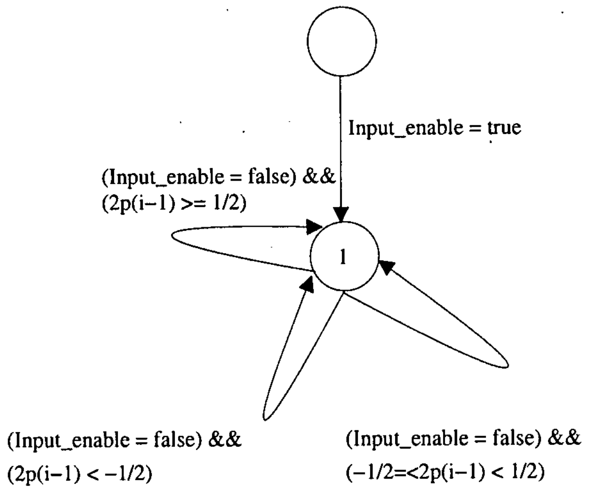

Input_enable = true
(Input_enable = false) &&
(2p(i−1) >= 1/2)
1
(Input_enable = false) &&
(2p(i−1) < −1/2)
(Input_enable = false) &&
(−1/2=<2p(i−1) < 1/2)

Figure 3.4: ACFG for SRT Algorithm. For brevity, only antecedents are shown.

```
-->P  := P - divisor            // --> one step of assignments
-->qp := qp + qp && qn := qn + qn && P:= P + P // left shift by one
-->qp := qp xor 1               // set quotient bit
-->P[2*N] == 0                  // if P is a positive number
   && remainder := P[2*N-1..N]
   && quotient := qp - qn
   ||
   P[2*N] ==1                   // if P is a negative number
   && remainder := P[2*N-1..N] + divisor
   && quotient := qp - qn - 1
```

Figure 3.5: A Example of ACFG Assignments. It is for the edge when $2P^{i-1} \geq 0.5$ in Figure 3.4. There are four ordered steps in this sequence assignment.

Figure 3.4 sketches the main part of the specification ACFG. The three self-loop edges correspond to the 3-way branch structure in the loop on lines 3–16 of Figure 3.3. The remainder and quotient are the outputs. They are the final results after N+1 cycle. However, because the specification is cycle-accurate, I check the intermediate results for each cycle by using the consequents in the ACFG.

A simple textual syntax makes it possible to write an ACFG model in a text file. I will use one edge to explain the syntax. The complete ACFG specification is in Appendix A.

Figure 3.5 is a code segment taken from the specification of the edge where $2P^{i-1} \geq 0.5$ in Figure 3.4. It is for lines 5 – 6 in Figure 3.3. P is the partial remainder. The variable *qp* is the bit to record that the next quotient bit gets an assignment 1; The variable *qn* is the case for -1. After each cycle, they all shift left by one bit. There are four ordered steps in this sequence assignment, indicated by the `-->` operator. Inside each step, the order of assignments doesn't matter.

Step 4 has predicated assignments. If the sign bit of P is 0 (positive),

```
   remainder := P[2*N-1..N]
&& quotient := qp - qn
```

will be executed. If not, then

```
      remainder := P[2*N-1..N] + divisor
   && quotient := qp - qn - 1
```

will be executed.

All experiments were run on a 2.6Ghz Pentium 4 with 4GB of RAM. When running with a reasonable BDD variable order, run time rather than memory was always the limiting factor. BDD-based model checking is extremely sensitive to the variable order. I have not found an obviously good variable order, and I feared that I might inadvertently bias my experiments if I chose a static order that is better for one method over the other. Instead, I ran my experiments with dynamic BDD variable reordering enabled, using CUDD_REORDER_SIFT in the CUDD BDD package. (I tried other reordering methods, e.g., CUDD_REORDER_SYMM_SIFT or CUDD_REORDER_ANNEALING. They showed similar trends, but with larger runtime and bigger BDD sizes.) The initial order was arbitrary, except that I put the variables that symbolically encode the graph (in the standard method without partitioning) at the top of the order. Doing so makes the initial BDD sizes similar for the standard method and my new partitioned method.[3]

Table 3.1 shows results for this experiment. BDD sizes varied, but were generally similar between the two methods, as expected. However, in all cases, the new partitioned model checking approach ran faster, sometimes much faster. Strangely, sometimes larger instances required less time and/or smaller BDDs. I believe this is due to dynamic reordering, and note that the dynamic reordering time dominated total runtime.

[3]In a paper at CAV 2004, Sebastiani et al. presented impressive results showing an exponential improvement in memory usage for GSTE-style partitioned model checking applied to conventional LTL model-checking [84]. Discussion at the conference, however, indicated that most or all of the memory usage improvement would disappear with a variable order that put at the top the BDD variables that symbolically encode the graph. My work partitions the model checking similarly to theirs, hence, my choice of initial variable order.

	Without Partitioning			ACFG Partitioning			
N	Time(s)	BDD size	reorder	Time(s)	BDD size	reorder	Speedup
7	12.27	5390	95.2%	6.99	7462	94.4%	1.76x
8	32.31	10307	95.7%	18.41	14635	95.6%	1.76x
9	21.89	8982	95.8%	14.97	9770	96.6%	1.46x
10	50.62	6619	74.9%	26.26	15566	93.1%	1.93x
11	392.66	10915	98.3%	379.52	59125	97.8%	1.03x
12	727.43	19722	98.1%	484.12	29792	98.0%	1.50x
13	1854.62	63555	95.9%	877.29	38756	97.1%	2.11x
14	950.25	22262	95.9%	633.86	44919	96.2%	1.50x
15	452.57	20036	97.6%	193.19	56982	96.4%	2.34x

Table 3.1: Unpartitioned Model Checking vs. ACFG Partitioned Model Checking. The results are with dynamic variable reordering. In all examples in this thesis, runtimes are measured using the system call "*gettimeofday*", which has microsecond-level resolution. Obviously, other factors (e.g, I/O, multitasking, clock accuracy, thermal effects, etc.) make measurement accuracy much less. For uniformity of presentation in this thesis, I round runtime to two digits after the decimal point for all results.

As an aside, I note that GSTE-style pruning using the antecedents is very helpful. If I complete the graph in Figure 3.4 with all the edges for when the antecedents are false as well as true, as would be needed in a standard CFG, the model checking is much more expensive (Table 3.2).

	ACFG Partitioning without Antecedent Pruning		
N	Time(s)	BDD size	reorder
7	25.33	6363	89.2%
8	40.49	6503	80.8%
9	3270	24481	91.8%
10	time out		
11	1461.98	9132	44.4%
12	time out		
13	time out		
14	time out		
15	time out		

Table 3.2: ACFG Partitioned Model Checking on Complete CFG. Run time limit is set to 1 hour. The model checking without antecedent pruning is much more expensive.

Chapter 4

Cutpoints for Embedded Software

This chapter is based on a paper published in the 5th ACM Conference on Embedded Software (EMSOFT'05) [36]. I give experimental results on applying cutpoint theory (see Section 2.4.3) to software verification. To be specific, I am focusing on formal equivalence checking of embedded software.

There were two reasons for me to try this theory first on the embedded software vs. embedded software problem instead of the software specification vs. hardware implementation problem, which is the main direction of my thesis. First, I already had implemented a prototype verification tool for the TI (Texas Instruments) C6x family of VLIW (Very Long Instruction Word) DSPs [34, 35], which was very handy to start my research. Second, there are some similarities between embedded software and the software specifications of hardware. Therefore, studying the equivalence checking of embedded software was a good starting point to gain a better understanding of the equivalence checking of software specifications vs. hardware implementations.

4.1 Introduction

Embedded software shares with hardware — and differs from desktop and enterprise software — the frequent need for extreme optimization. The software must hit hard performance, power consumption, and code-size targets. Code that is slightly too big might necessitate moving to a larger, more expensive device, or code that is slightly too slow might result in unacceptable, non-real-time performance. Therefore, very aggressive optimization is the norm, including possible manual tuning of synthesis(hardware)/compiler(software) output. Compounding the problem, the underlying embedded processor is often designed with similar optimization goals — maximum performance at lowest cost or power, with minimal consideration to the ease of writing or understanding code. The instruction set of embedded processors (including DSPs) often are highly non-orthogonal, have many specialized instructions, and perform many operations in parallel, with the resulting artifacts (exposed pipelines, long branch delays, VLIW, etc.). All of these features enable very highly optimized, high-performance code, but they also greatly complicate code generation and optimization. Finally, the embedded market is less tolerant of defective software than some other software markets, because patching embedded software in the field can be too difficult, too expensive, or unacceptable to customers. All of these factors point toward very demanding verification requirements. I focus on a particular verification problem: verifying the functional equivalence of two similar segments of low-level code, as would be needed, for example, after hand-tuning compiler-generated code.

Inspired by the success of applying cutpoint theory (mentioned in Section 2.4.3) into formal equivalence verification of combinational hardware, I introduce cutpoint-style analysis into the formal equivalence verification of embedded software. I have implemented the ideas in my proof-of-concept verification tool targeting the Texas Instruments C6x family of VLIW DSPs. My experimental results show large improvements in memory usage and runtime over earlier methods.

4.2 Related Work

Automatic formal verification of software has been enjoying a renaissance lately, with much of the focus on extending finite-state model checking [24] — which has been successfully applied to sequential circuits, protocols, and other reactive systems — to software, viewed at a system-level as a reactive (non-terminating) system (e.g., [6, 47, 94]).

A complementary line of work, more relevant to my work, has focused on formally verifying the equivalence of low-level code, e.g., to higher-level specifications [93, 3, 46], to other versions of low-level code [30, 35], or to hardware [22, 81]. In the compiler-research community, the work of validation of compiler optimizations [82, 74, 100] is also relevant to my work.

The above research of equivalence checking typically verifies a relatively small segment of code as a transformational rather than reactive system, i.e., the code computes a result and terminates, analogous to a combinational circuit in hardware. The basic approach is to use symbolic execution of the code to compute the formal relationship between inputs and outputs, and then prove that the outputs are always equivalent. The lower-level emphasis is well-suited for the verification of optimizations needed for embedded systems. For example, Currie and Hu have demonstrated this approach successfully verifying (or finding bugs in) code optimizations for complex embedded processors [30]. Unfortunately, the basic approach is not scalable: the representation of the input/output relationship or the complexity of deciding equivalence blows up in memory, runtime, or both. In one embarrassing example, a 47-line assembly language routine required 15 hours to verify [30]! (Granted, the dynamic instruction count after loop-unrolling was a few thousand instructions, and the verification would have run much faster had certain expensive, but unnecessary, optimizations been disabled.)

4.3 Basic Verification Approach

I briefly mentioned some general approaches for symbolic simulation in Section 2.4.2. In this section, I focus on a specific application of symbolic simulation — formal equivalence verification of embedded software. I will first give a domain-specific review of this problem. More extensive introductions are available elsewhere (e.g., [11, 29]).

The verification task is to take two assembly-language routines that compute some values and terminate, and verify that they are equivalent. The user specifies what inputs are initially equal and what outputs should be equal when the routines terminate. The assumption is that the two routines have very similar control-flow (basically, along corresponding paths of two routines, the branches that are encountered are equivalent. See [29, p. 67] for details.). If this assumption is violated, the verifier might declare inequivalent two routines that actually compute the same value, but it will not claim equivalence for two routines that are not. As in previous work [30, 35, 29], some additional simplifying assumptions are needed (e.g., no self-modifying code, no recursion, etc.). I do not repeat them here.

The verification procedure requires a simple model of the processor at the instruction set architecture level, and then uses this model to simulate the two routines. However, instead of computing actual values, the simulator is symbolic and computes expressions that denote the values as a function of the initial inputs and states. For example, consider the following (TI C6200) code segment:

```
ADD .L2 B1, B0, B2  ; B2:=B1+B0
ADD .L2 B2, B0, B3  ; B3:=B2+B0
```

If I denote the initial values of registers B0, B1, and B2 as $B0_0$, $B1_0$, and $B2_0$, then after these two instructions execute, the simulator will compute the "values" in registers B2 and B3 to be symbolic expressions "$B1_0+B0_0$" and "$(B1_0+B0_0)+B0_0$".

I call the above style of symbolic simulation *functional translation*, because it computes the values at each point as a function of the initial values. An alternative, which I call *relational translation*, computes for each instruction a clause that relates the values before and after execution. For example, for the above code, I would generate the relation $(B2_1 = B1_0+B0_0)\wedge(B3_1 = B2_1+B0_0)$. The functional translation can have worst-case exponentially-sized expressions; the relational translation guarantees an expression size linear in the length of the execution sequence, but at the cost of many more variables, which blows up the complexity of deciding equivalence. Others have argued for the superiority of the relational translation [11]; I will revisit this issue later.

To keep the equivalence of symbolic expressions decidable, only constant propagation and linear arithmetic (i.e., symbolic expressions can be added together and multiplied by constants) are interpreted. More complex operations (e.g., multiplication of symbolic expressions, or any arbitrarily complex operations) are treated as *uninterpreted functions* (See Section 2.5.1). This abstraction hides datapath complexity and is safe, but sometimes too conservative — being unable to prove the equivalence of a shift and a divide-by-2, for example — so additional domain-specific rewriting rules are needed to handle those cases. I also use special interpreted functions `read` — which given a memory and an address, denotes the value at that address — and `write` — which given a memory, an address, and a value, denotes an updated memory in which the value has been written to the address. The key axiom is that

$$\text{read}(\text{write}(m, a1, v), a2) = \begin{cases} v & \text{if } a1 = a2 \\ \text{read}(m, a2) & \text{otherwise} \end{cases}$$

To successfully verify low-level code, I have found it necessary to model memory layout accurately. In particular, when verifying software written in a high-level language, arrays are often assumed to be disjoint, so the read/write functions can be applied to each array separately (e.g., a write to an array A does not change the state of array B). In contrast, I model all of memory (or each bank of memory in a system with multiple banks) as a single array with all reads and writes directed at this array. This approach accurately models low-level memory layout, such as relative addressing (e.g., address arithmetic performed on index registers, based on knowing the layout of data in memory). However, the approach leads to large symbolic expressions, so I rely on some rewriting optimizations to try to keep expression size manageable [30, 29]. Efficient decision procedures exist for this combined logic (linear arithmetic, uninterpreted functions, and read/write); I use the Stanford Validity Checker (SVC) [7].

I use simple techniques to handle control flow. These techniques have proven adequate to handle the bottom-level, highly-optimized computational kernels I am targeting. For backward branches, the analysis essentially unrolls loops: if the decision procedure can prove that the branch is taken, the tool takes the branch; if the decision procedure can prove that the branch is not taken, the tool doesn't take it; otherwise, the tool declares that the code contains branching that it can not handle and gives up. All fixed-iteration loops, which are the common case in low-level DSP code, can be handled this way. For forward branches, the tool also first tries to prove the branch as taken or not taken. Otherwise, it case-splits. Based on my assumption that the two routines being compared have similar control structure, I require that the two routines encounter "compatible" forward branches in the same order: the two branches must always branch the same way or always branch opposite ways (to allow reordering taken/not-taken paths). With compatible branches, my tool proceeds to verify that the routines are equivalent along both paths. With incompatible branches, my tool declares that it cannot verify the routines equivalent. In the C6x family, all instructions are predicated, so I rarely encounter forward branches.

Overall, I have found it straightforward to build symbolic simulators, even for complex processors. The basic verification approach works well on small, intricately optimized code segments. However, as mentioned earlier, the basic approach does not scale well to longer segments of code.

4.4 Cutpoints for Software

Analogously to formal equivalence verification of combinational circuits, I would like to use cutpoints to gain scalability. Obviously, the method needs to be conservative — it should not declare equivalence when the code segments are not equivalent — but I must also avoid introducing too many false inequivalences.

The most fundamental question is how to define cutpoints for software. In a combinational circuit, values flow along wires, from the inputs to the outputs, with gates performing computation along the way. Similarly, in software, values flow through the code in the program state (variables for high-level software; registers, internal buffers, memory, and other machine state for low-level software), with each instruction performing some computation on the values as they pass by. Thus, a cutpoint in software is some part of the program state at some point in a program that is provably equal to some part of the program state at some point in the other program. In a combinational circuit, a verification tool can ignore the logic driving the cutpoint and insert a new primary input. Instead, for software, a tool can discard the symbolic expression computed for the value at the cutpoint and replace it with a new symbolic variable. If the tool can verify equivalence using the cutpoint, then the original circuits or programs were equivalent.

Control flow adds a wrinkle to the above definition. In combinational hardware, every wire always has a value for every possible input value. In software, some instructions may never be executed for some input values, and other instructions may be executed multiple times. Therefore, the value of the program state at a given point in a program isn't always well-defined. For example, a variable at a given point has no value if the point hasn't been reached; alternatively, if a path revisits the point, the variable could have different values each time. The solution is to define software cutpoints *dynamically*, based on each dynamic execution path, rather than on the static code.

Definition 4.1 (Software Cutpoint) *A cutpoint in software is some part of the program state at some point on an execution path in a program, which is provably equal to some part of the program state at some point on an execution path in the other program.*

I will use the existing verification approach to enumerate paths and attempt to use cutpoints to make the verification of each path much more efficient.

Example: Consider a short loop that zeroes out a 1024-word block of memory. As a simple example, I will verify the equivalence of the loop to itself. Unrolling the loop, the instruction stream is simply 1024 store instructions:

```
STW .D1 A0, *A4++
STW .D1 A0, *A4++
...
STW .D1 A0, *A4++
```

where register A0 has been initialized to 0, and register A4 indexes through the memory block using auto-increment. Using my basic verification approach, after i iterations, the symbolic expression for memory will be:

$$\text{write}(\ldots\text{write}(\text{write}(m_0, A4_0, 0), A4_0 + 4, 0)\ldots, A4_0 + 4(i-1), 0)$$

where m_0 and $A4_0$ are the initial values of memory and register A4. The expression grows linearly with iteration count. Using the relational translation would also give linear-size expressions (linear number of constant-size clauses), plus a linear number of new variables. However, if I use cutpoints, I find that the machine states of the "two" programs (the two copies of itself) agree completely after each instruction. Hence, after each instruction, I could introduce a new cutpoint memory variable m_i and a new cutpoint address value $A4_i$, and then at the next instruction have to prove only the equivalence of $\text{write}(m_i, A4_i, 0)$ in the two code segments.

The above example is contrived, but it serves to highlight the key design decisions in trying to apply cutpoints to software:

- *Where and how fine-grained to look for cutpoints?* In the above example, after every instruction, the entire machine state matched between the programs being compared, so I could cut the entire state between instructions. That would work for the simple example, but would produce false inequivalence for anything non-trivial. On the other extreme, I could try to match each register and memory location, or even each bit of each register and memory location for interpreted values at each instruction, as a possible cutpoint. Finer-grained cutpoints allow more flexibility and improve the possibility of matches, but they result in an exploding set of possible matches to be considered. Also, I may not want to look for or insert cutpoints for some parts of the state: for example, in the simple example, if I make the loop induction variable a cutpoint, I lose the ability to prove termination.

- *How to find cutpoints?* In the simple example, the two programs were synchronized in lock-step, so I was able to execute a single instruction from each and find matching cutpoints. In general, however, computations will be reordered and instructions will be optimized away, so I need techniques to look for possible cutpoints.

- *Whether to do the cut?* This is the dynamic version of the first question. Once I find (and prove) a cutpoint, I may choose not to use it, perhaps to avoid false inequivalences. A verification tool would need a heuristic to make this decision.

- *How to do the cut?* By definition, I create a new symbolic variable to take the place of the expression computed for the cutpoint. But how aggressively do I propagate this new cutpoint variable? By the time the tool discovers a cutpoint, the symbolic simulator may have already computed other symbolic expressions, for other parts of the machine state, based on the symbolic expression being cut out. Should I track down these dependent expressions? How?

- *How to reduce false inequivalences?* The previous questions will affect the false negative rate, but this question is important enough to consider independently. Should I add constraints on newly introduced cutpoint variables? How do I save those constraints. Are there other ways to reduce false inequivalences?

Any implementation of software cutpoints must answer the above questions. Ultimately, the real question is "Do the answers to the above questions allow verifying real code more efficiently and with an acceptable level of false inequivalences?"

4.5 Proof-of-Concept Implementation

To test the effectiveness of software cutpoints, I have implemented an instance of the idea. My proof-of-concept implementation is built on top of my existing tool, which uses the basic verification approach from Section 4.3 and targets assembly code for the Texas Instruments C6x family of VLIW DSPs [35]. In this section, I discuss my answers to the design questions raised above.

Where and how fine-grained to look for cutpoints? I check the symbolic expressions for only the memory, and treat the entire memory as a single possible cutpoint.

I do not look for cutpoints between registers or other parts of the machine state due to the following three reasons. First, the embedded software examples being verified compute and save results into memory, then terminate. Such programs use registers only to storage temporary results. The equivalence of such two programs is proven by checking the memory contents after they terminate. Second, if the verification algorithm inserts cutpoints for registers, it is possible that loop induction variables can be cut off. Therefore, the symbolic simulation might not terminate any more. Third, my experience indicated that the symbolic expressions for memory are the primary source of blow-up in the basic verification approach.

I treat the entire memory as a single possible cutpoint when looking for cutpoints for memory expressions, i.e. during symbolic simulation, only when two embedded software programs have equivalent symbolic expressions for the whole memory, does my algorithm look for and insert cutpoints. Such coarse-grained cutpoints can greatly simplify the task of searching for cutpoints — the verification algorithm doesn't need to find cutpoints for specific memory locations.

How to find cutpoints? Checking only the entire memory makes this task much easier. The symbolic expression for memory changes only after a store instruction, so I keep a history buffer of the memory expressions from the last k stores, for some depth k. The verification tool simulates one program through k stores, then the other program through k stores, then calls the decision procedure to find the most recent match (if any) of the k^2 possibilities.

A large value of k can save time on decision procedure calls. For example, consider two programs P_1 and P_2 being verified. Program P_1 has one order of computations (each computation stores results into memory), c_0, c_1, c_2, c_3, c_4, c_5, ..., c_{2n}, c_{2n+1}. Program P_2 has another ordering, c_1, c_0, c_3, c_2, c_5, c_4, ..., c_{2n+1}, c_{2n}. A buffer size of $k = 4$ reduces the number of decision procedure calls by half compared to a buffer size of $k = 2$, because the tool compares memory only after every fourth store.

However, k should not be too large. Otherwise, it can slow down the tool due to the following reasons. First, the cost of saving the history buffer is not trivial. Second, in order to find a most resent match, the tool has to check the two history buffers of the two programs. The worst case complexity is $O(k^2)$ decision procedure calls, which is expensive for large k. Third, the memory expression can grow exponentially with k.

Consequently, I informally tried several values of k, and chose $k = 10$ for all of my experiments.

Whether to do the cut? When I find a cutpoint, I always do the cut. Again, this is motivated because memory expressions tend to blow up, and because I am matching only memory, so loop induction variables in registers won't be cut.

How to do the cut? I could conceivably match a memory expression k stores earlier, which could be an unbounded number of (non-store) instructions in the past. It's hard to imagine trying to compute directly the effect of introducing the cut variable on all the values (and control flow!) that may have been computed subsequently. Furthermore, the C6x family has very deep pipelines, so searching through all the symbolic expressions in the pipeline and reasoning about any pipeline interactions is a daunting task. Instead, I simply leverage the fact that I already have a symbolic simulator for the processor. After each store instruction, I record in the history buffer the entire machine state, not just the expression for memory. When I find a cutpoint, I roll back the simulation to the cutpoint, and re-simulate any subsequent instructions.

How to reduce false inequivalences? This is the most complex question to answer. The simplest answer is to do nothing special. I implemented that choice and found that it worked successfully, and very efficiently, on some examples (e.g., the industrial-inspired "Hup" example in Section 4.6), but produced too many false inequivalences (e.g., on the software pipelining example in Section 4.6). The fundamental problem is the inability to handle reordering of independent memory accesses. For example, consider verifying

```
LDW .D1 *A3++, A1
NOP 4  ; 4 cycle NOP for load to complete
STW .D1 A0, *A4++
```

versus

```
STW .D1 A0, *A4++
LDW .D1 *A3++, A1
NOP 4  ; 4 cycle NOP for load to complete
```

If I know that registers A3 and A4 point to different locations, then the two code segments are equivalent, and the basic verification approach would successfully verify that. Using my simple cutpoint approach, however, I would introduce a cutpoint after the STW instructions. The value of A1 at the end of the first code segment, therefore, will be based on the pre-cutpoint memory expression, e.g., read($m_{\text{old}}, A3_0$), whereas the value of A1 at the end of the second code segment will be based on the post-cutpoint memory expression, e.g., read($m_{\text{new}}, A3_0$), which aren't equivalent. The verification returns a false inequivalence.

I introduced two ways to eliminate these false inequivalences. I call the first "memory look-through". In this approach, the verification tool records the address written for every store instruction. For each load instruction, the tool tries to prove the independence of the address being read from the addresses that have been written. The read expression that is generated can read from any version of memory back to the most recent store that cannot be proven independent of the address being read. For my implementation, it turned out to be faster to first ask SVC to prove that the load address is independent of *all* stores, in a single decision procedure call. If this succeeds, the read expression reads from the initial memory. Otherwise, the read expression reads from the most recent store that cannot be proven independent of the address being read. In the example above, if the address ranges for A3 and A4 provably never overlap, then the value loaded into A1 will always be read($m_0, A3_i$), where m_0 is the initial memory. This method reduces, but does not eliminate false inequivalences.

The other approach I tried completely eliminates false inequivalence (from the cutpoints — obviously, false inequivalence from other aspects of the verification approach, such as uninterpreted functions, remain). When a new cutpoint variable is introduced, I add an assertion to the decision procedure that the new cutpoint variable is equal to one of the two (proven equivalent) expressions that it is replacing. This assertion guarantees that the cutpoint variable will always be properly constrained. I call this approach "memory assertions", and it is analogous to the combinational circuit equivalence technique in which, rather than introducing a new primary input at the cutpoint, I simply drive the cutpoints in both circuits from the same circuitry in one [12].

With plausible answers to all the design decisions, I can proceed to the real question: does it work on real code?

4.6 Experimental Results

I have run experiments using several test cases. They are all small computational kernels that performs computations over arrays, where I can scale the difficulty of the example by adjusting the loop count. In each case, I verified the equivalence of unoptimized and highly optimized versions of the code. I compare the performance of four different methods: the basic verification approach, using the functional translation; the basic verification approach, using the relational translation; the functional translation with cutpoints, using memory look-through; and the functional translation with cutpoints, using memory assertions. For the functional translation methods, I use memory rewriting optimizations to try to reduce blowup [30, 29], but I enable only the rewrites that actually help in the examples. Doing so helps the basic functional translation, but makes no difference for the cutpoint methods. This biases the experiments against the cutpoint methods.

```
(... linkage and initialization omitted.
     BO is the loop counter ...)
   L12:      ; PIPED LOOP KERNEL
         LDW    .D2    *B5++,B4
   ||    LDW    .D1    *A3++,AO
         NOP    2
   [ BO]SUB     .L2    BO,1,BO
   [ BO]B       .S2    L12
         MPYSP  .M1X   B4,AO,AO
         NOP    3
         STW    .D1    AO,*A4++
(... subroutine return omitted ...)
```

Figure 4.1: Unpipelined Assembly Code. The vertical bars indicate instructions executed in parallel. `LDW` (load word) has 4 delay slots, branches have 5 delay slots, and `MPYSP` (single-precision multiply) has 3 delay slots. The code point-wise multiplies two arrays, storing the result in a third array. The code takes 10 cycles per iteration. (Listing taken from [75].)

The first test case is taken from an article on DSP code optimization, explaining how to optimize code for high-performance DSPs [75]. The example demonstrates software pipelining a short loop, targeting the C67x. Software pipelining is a powerful instruction scheduling technique that exposes additional parallelism in loops, thereby improving performance [63]. The basic idea of software pipelining is to rearrange the computation such that portions of different loop iterations execute at once, similarly to hardware pipelining. A prologue is required to start the pipelined computation, and an epilogue is required to "flush the pipeline" at the end of the computation. Figure 4.1 shows the unpipelined code, and Figure 4.2 gives the software pipelined code. The task is to verify the equivalence of the two.

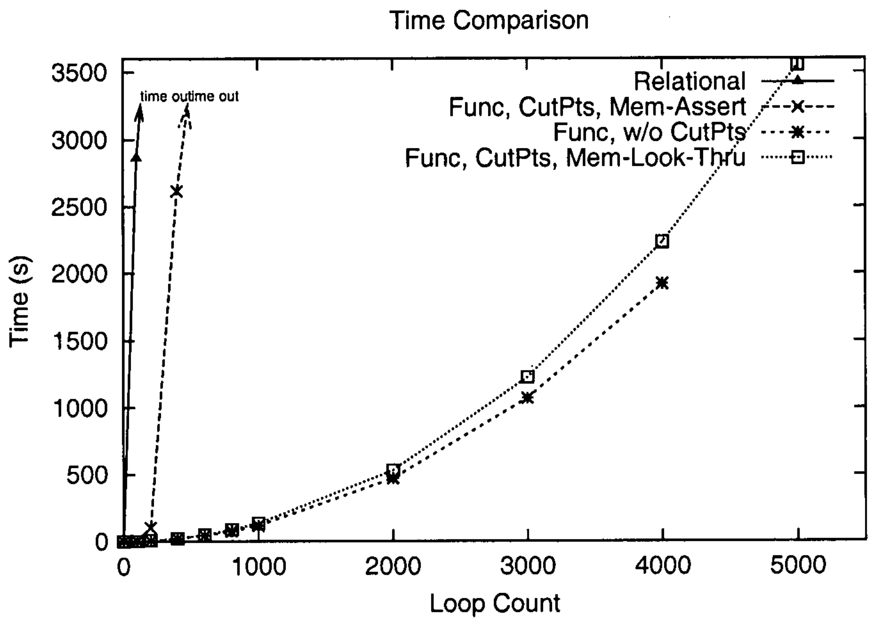
```
(...linkage & initialization omitted.
    B0 is the loop counter ...)
L8:     ; PIPED LOOP PROLOG

        LDW   .D2   *B5++,B4 ;
||      LDW   .D1   *A3++,A0 ;

        NOP   1

        LDW   .D2   *B5++,B4 ;@
||      LDW   .D1   *A3++,A0 ;@

   [B0] SUB   .L2   B0,1,B0  ;

   [B0] B     .S2   L9       ;
||      LDW   .D2   *B5++,B4 ;@@
||      LDW   .D1   *A3++,A0 ;@@

        MPYSP .M1X  B4,A0,A5 ;
|| [B0] SUB   .L2   B0,1,B0  ;@

   [B0] B     .S2   L9       ;@
||      LDW   .D2   *B5++,B4 ;@@@
||      LDW   .D1   *A3++,A0 ;@@@

        MPYSP .M1X  B4,A0,A5 ;@
|| [B0] SUB   .L2   B0,1,B0  ;@@

;** ----------------------------*
L9:     ; PIPED LOOP KERNEL

   [B0] B     .S2   L9       ;@@
||      LDW   .D2   *B5++,B4 ;@@@@
||      LDW   .D1   *A3++,A0 ;@@@@

        STW   .D1   A5,*A4++ ;
||      MPYSP .M1X  B4,A0,A5 ;@@
|| [B0] SUB   .L2   B0,1,B0  ;@@@

;** ----------------------------*
L10:    ; PIPED LOOP EPILOG
        NOP   1

        STW   .D1   A5,*A4++ ;@
||      MPYSP .M1X  B4,A0,A5 ;@@@

        NOP   1

        STW   .D1   A5,*A4++ ;@@
||      MPYSP .M1X  B4,A0,A5 ;@@@@
        NOP   1
        STW   .D1   A5,*A4++ ;@@@
        NOP   1
        STW   .D1   A5,*A4++ ;@@@@
(... subroutine return omitted ...)
```

Figure 4.2: Software Pipelined Assembly Code. If the inputs are declared to be `const`, the compiler does software pipelining, improving performance to 2 cycles per iteration. But, does this do the same thing as Figure 4.1? (Listing taken from [75].)

	Functional w/o Cutpoints		Functional with Cutpoints	
Loop Count	Time(s)	Memory(MB)	Time(s)	Memory(MB)
200	6.13	10.6	6.29	5.3
400	24.32	27.1	24.49	6.0
600	54.50	53.7	54.65	6.7
800	97.22	90.0	96.84	7.6
1000	149.96	132	150.13	8.7
2000	600.98	513	596.47	13.7
3000	1425.63	1150	1363.85	23.4
4000	2461.32	2023	2490.58	27.6
5000		mem out	3939.21	29.3

Table 4.1: Software Pipeline Detailed Results. Memory limit is set to 2 GB.

I had previously been able to verify this example, using the basic verification approach (without cutpoints). Using cutpoints, I was still able to verify equivalence of the two versions, taken unmodified from the article. Because the cutpoint methods are conservative, successfully proving equivalence shows the cutpoints were sufficiently accurate and did not create false inequivalences. Figure 4.3 shows the performance trends as I scaled the number of loop iterations. The relational translation performs strikingly poorly: the run time blows up immediately, but surprisingly, so does the memory usage. Apparently, the theoretically linear expression size growth of the relational translation is not competitive with the savings possible with the memory rewriting optimizations of the functional translation. The method using cutpoints and memory assertions also times out on small instances. Apparently, forcing the decision procedure to reason about all the cutpoint variables is causing blow-up, similar to the relational approach. Perhaps a newer decision procedure would help, as SVC is several years old. Nonetheless, on this example, cutpoints provide a vast improvement in memory usage at a small cost in run time.



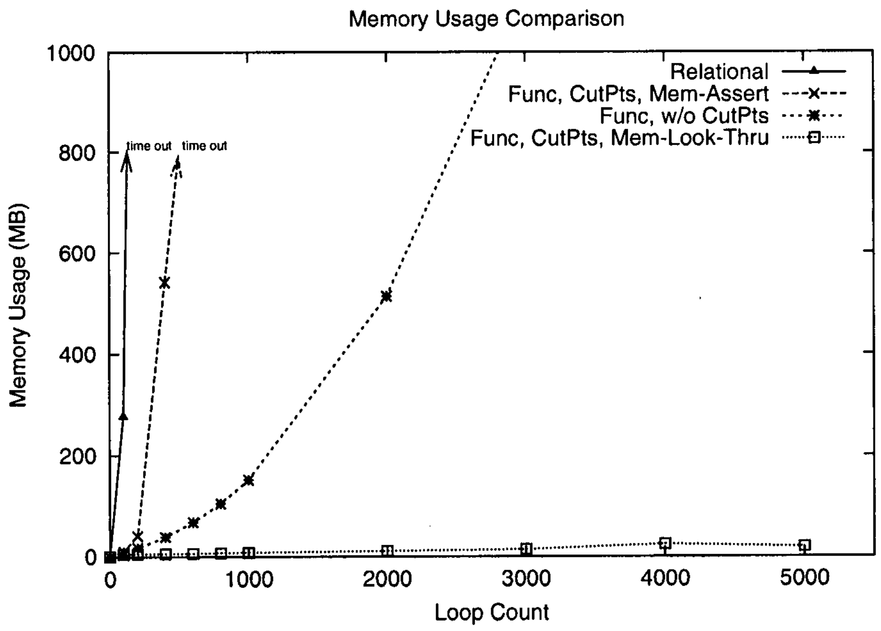


Figure 4.3: Software Pipeline Results. The relational translation times out even for minuscule numbers of iterations, and the functional translation with cutpoints and memory assertions times out quickly, too. My previous functional translation method without cutpoints is fastest, but the memory usage blows up. The new cutpoint method with memory look-through is almost as fast and uses very little memory. Run time limit is set to 1 hour and memory limit is set to 2 GB

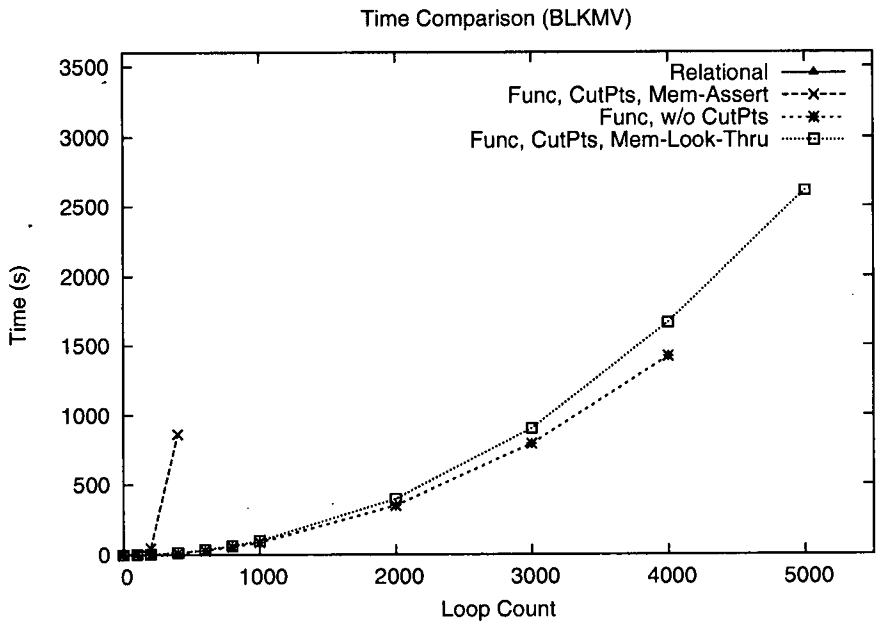
The preceding experiment used compiler-optimized code. For a harder experiment, I ran experiments on expert, hand-optimized code. Texas Instruments provides the TMS320C67x DSP Library (DSPLIB), a freely downloadable library of commonly-used DSP signal-processing routines hand-tuned by TI experts to achieve optimal execution speed [92]. Furthermore, each library function includes an equivalent C reference model, which I can compile using TI's TMS320C6x ANSI C compiler to get an equivalent, non-optimized version. For my experiments, I selected three simple routines with numerous memory writes (the main source of expression size blow-up) from the library: block move (`DSPF_sp_blk_move`, 43 lines of code), convolution (`DSPF_sp_convol`, 101 lines of code), and FIR filtering (`DSPF_sp_fir_r2`, 270 lines of code).

On my initial attempt to verify these examples, the cutpoint methods failed immediately with false inequivalences. The problem is that the hand-tuned code's subroutine linkage is different from the compiler-generated code: caller state is saved and restored slightly differently. This highlights the immaturity of my initial heuristics — a small change was able to foil my cutpoint implementation. More sophisticated heuristics will obviously be needed in practice.

Fortunately, the subroutine linkage code was easy to remove, so I was able to try the verification (expert-hand-tuned vs. compiler-generated) on only the computational kernels of each routine. I manually replaced the linkage code with NOPs, added assertions to the decision procedure to initialize the two routines in the same way, and re-ran the verification tool. This time, I was able to verify equivalence fully automatically, demonstrating that my cutpoint heuristics were accurate enough for the computational kernels of my test cases. Figures 4.4, 4.5, and 4.6 show the performance trends for these examples. On the block move example, the performance is very similar to the software pipelining example: the relational translation times out immediately; the cutpoints method with memory assertions times out quickly, too; the basic functional translation is fastest, but blows up in memory; and the cutpoint method with memory look-through is almost as fast and doesn't suffer memory blow up. On the convolution and FIR examples, though, the results are even better: now, the cutpoint method with memory look-through is roughly twice as fast as the basic functional translation. The reason for this performance difference appears to be that in the hand-optimized convolution and FIR examples, the computation is much more highly reordered, resulting in a much harder equivalence expression for the decision procedure. The overhead of finding cutpoints is swamped by the savings of a simpler final verification problem. To test this hypothesis, I ran experiments using larger convolutions and more FIR filter coefficients and found that the performance advantage of the cutpoint method increased. Conversely, the block move example has no computation at all, simplifying the final verification problem, so the relative overhead of the cutpoints is higher.

In all the test cases, the cutpoint method with memory look-through vastly reduced memory usage. Run time ranged from a minor increase to a significant decrease. Accuracy was good enough to verify all of the computational kernels. Clearly, cutpoints can be very effective.

Tables 4.1, 4.2, 4.3, and 4.4 give detailed results comparing the two competitive methods: my original basic verification approach, without cutpoints, using the functional translation and memory rewriting, and my new cutpoint-based method, using the functional translation and memory look-through. All experiments were run on a 2.6Ghz Pentium 4 with 4GB of RAM. I set the run time limit to 1 hour.



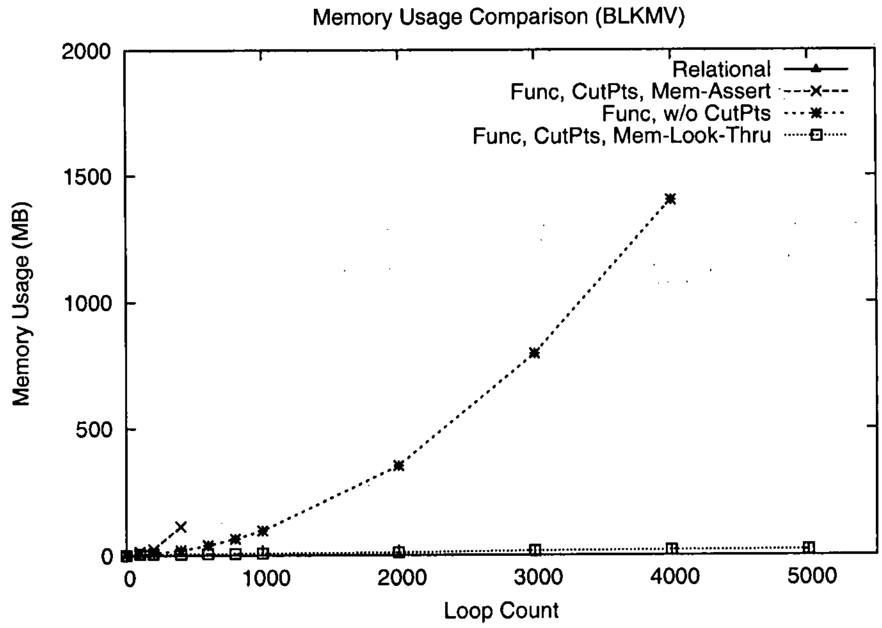
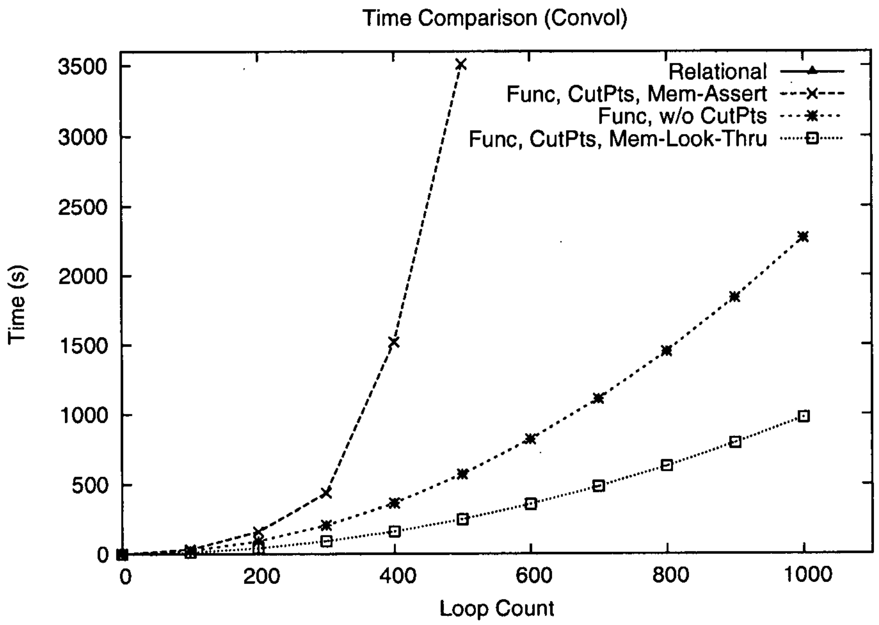


Figure 4.4: Block Move Results. Performance trends are similar to Fig. 4.3, except that the time overhead of cutpoints is larger. The relational translation times out at the very beginning and is not shown in this figure. Run time limit is set to 1 hour and memory limit is set to 2 GB.



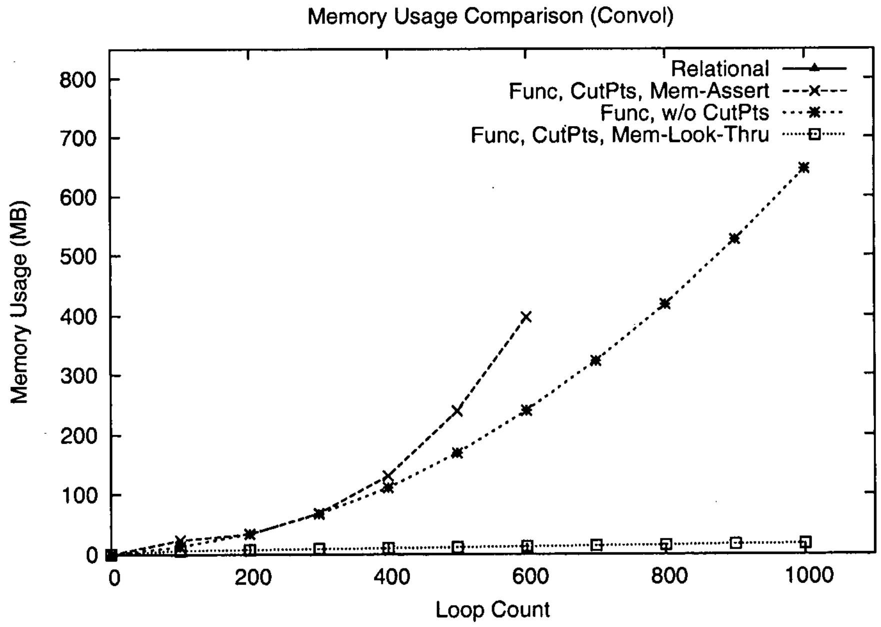
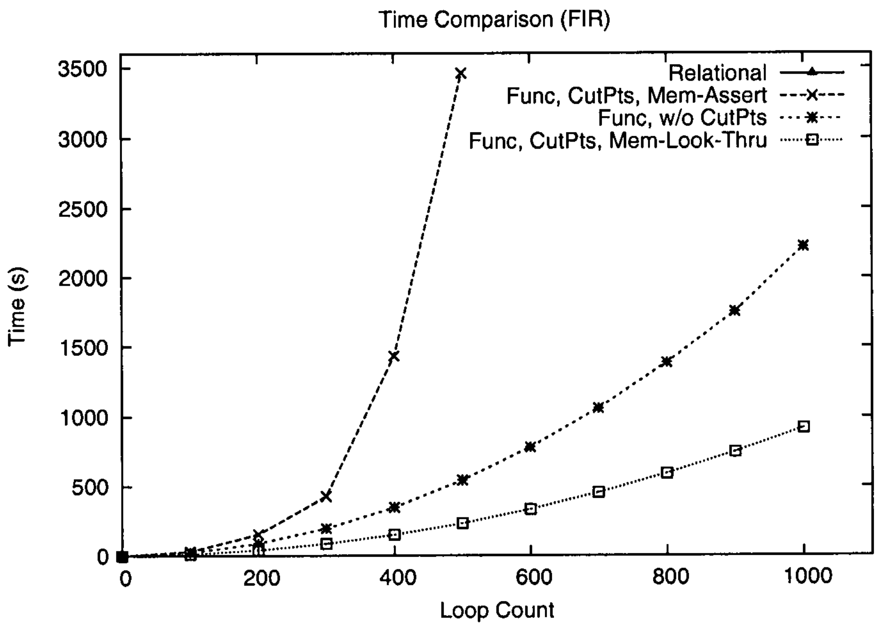


Figure 4.5: Convolution Results. Here, not only does the cutpoint method with memory look-through have the lowest memory consumption, but it is fastest, too. The relational translation times out at the very beginning and is not shown in this figure. The runs were with the number of impulse response samples set to 8. Run time limit is set to 1 hour and memory limit is set to 2 GB.



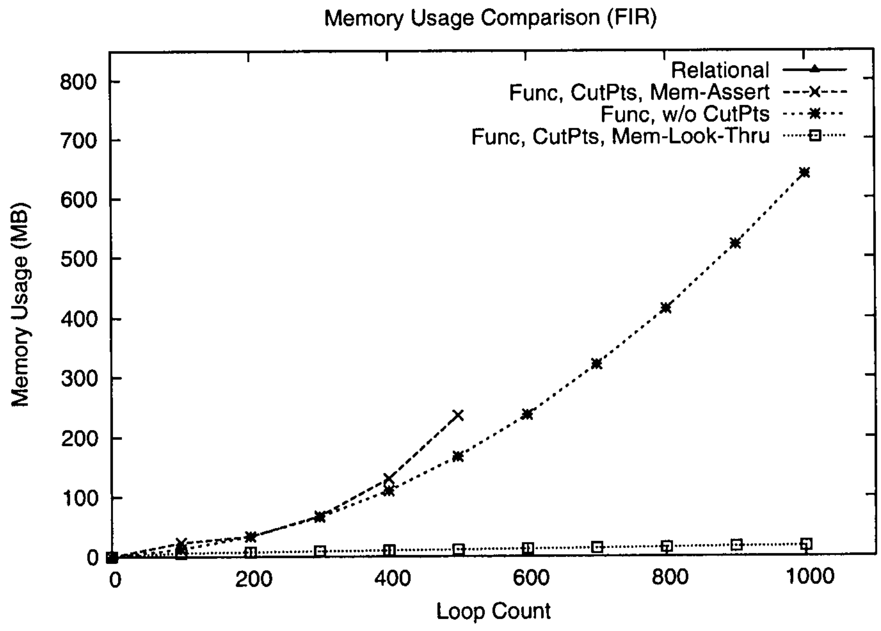


Figure 4.6: FIR Filter Results. Performance trends are similar to Fig. 4.5: cutpoints with memory look-through was fastest and used vastly less memory. The relational translation times out at the very beginning and is not shown in this figure. I set the number of filter coefficients to 8. Run time limit is set to 1 hour and memory limit is set to 2 GB.

	Functional w/o Cutpoints		Functional with Cutpoints	
Loop Count	Time(s)	Memory(MB)	Time(s)	Memory(MB)
200	3.59	8.7	4.15	5.0
400	14.05	20.3	15.99	5.6
600	31.53	39.1	35.90	6.1
800	56.00	64.6	63.65	6.7
1000	87.22	96.5	99.04	7.3
2000	349.06	353	395.70	10.2
3000	793.32	796	903.46	16.1
4000	1420.66	1401	1665.15	19.3
5000		mem out	2615.88	21.0

Table 4.2: Block Move Detailed Results. Run time limit is set to 1 hour and memory limit is set to 2 GB.

	Functional w/o Cutpoints		Functional with Cutpoints	
Loop Count	Time(s)	Memory(MB)	Time(s)	Memory(MB)
100	23.32	12.5	11.34	6.0
200	91.64	33.7	41.96	7.2
300	204.96	67.1	91.42	8.5
400	363.38	111	159.91	9.7
500	569.05	169	247.60	10.9
600	818.84	240	354.77	12.2
700	1109.54	323	481.52	13.4
800	1451.09	418	627.45	14.6
900	1836.78	526	792.95	15.9
1000	2267.42	646	976.55	17.1

Table 4.3: Convolution Detailed Results

	Functional w/o Cutpoints		Functional with Cutpoints	
Loop Count	Time(s)	Memory(MB)	Time(s)	Memory(MB)
100	22.83	12.4	10.72	6.0
200	88.01	33.6	39.38	7.3
300	195.28	66.4	85.21	8.6
400	345.95	110	149.41	9.7
500	539.08	167	231.48	11.0
600	776.26	237	332.31	12.2
700	1058.00	320	451.26	13.4
800	1381.92	414	587.61	14.6
900	1752.31	521	742.46	15.9
1000	2216.26	640	915.77	17.1

Table 4.4: FIR Filter Detailed Results

Chapter 5

Cutpoints for Software vs. Combinational Circuits

Motivated by my adaptation of the cutpoint idea to software, I return to the original problem — equivalence verification of software specifications vs. hardware implementations. In this chapter, which is based on my paper published in the 43rd Design Automation Conference (DAC'06) [37], I will give the early cutpoint insertion approach for equivalence checking of software specifications vs. combinational circuits.

5.1 Introduction

One approach to verifying a hardware implementation against a software specification is first to convert the high-level software into RTL or gate-level hardware, and then to leverage standard techniques from RTL or gate-level combinational equivalence verification — in particular, the introduction of cutpoints — to verify the equivalence of the two low-level models. Unfortunately, as Gupta et al. [44] point out, most high-level synthesis has targeted creating multicycle, resource-constrained designs, rather than creating single-cycle hardware from software specifications with complex control flow. As a result, off-the-shelf high-level synthesis isn't applicable to my problem.

Instead of high-level synthesis, the alternative is symbolic simulation. With symbolic simulation, the challenge is complexity blow-up. To handle the control flow of software, symbolic simulation must explore all paths through the code. If this exploration is done explicitly, execution time can blow up with the exponential number of paths. If paths are merged to avoid this blow-up, then there is a potential blow-up in the expressions that track the different results that are computed on the different paths.

The principle contribution of my early cutpoint insertion approach is a novel way to introduce cutpoints *early*, during the analysis of the software specification, rather than after a low-level hardware-equivalent has been generated. By doing so, I avoid the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. I evaluate my method on a challenge problem suggested to us by colleagues in industry: a family of instruction length decoders for varying subsets of the IA-32 instruction set architecture. Experimental results show large reductions in runtime and memory usage due to my early insertion of cutpoints.

5.2 Related Work

Unlike most high-level synthesis systems, Gupta et al.'s Spark system was specifically designed for the sorts of high-level models I need (highly unoptimized, complex software into single-cycle hardware) [44]. Indeed, they demonstrated the capabilities of their system on a IA-32 instruction length decoder, the same challenge problem I use. Their decoder is much simpler than mine, but with the same essential characteristics. Unfortunately, the current version of Spark was not able to handle my examples, so I cannot evaluate how well this approach would work.[1] In my initial software analysis phase, I assume that certain standard program analyses (e.g.,

[1]Spark v1.2, released Feb 5, 2004. It would generate only multi-cycle implementations for my examples. Even the tool's author could not get it to generate a combinational circuit implementation.

CFG construction, data flow analyses) have been done. I believe optimizations and analyses as done in Spark could enhance the initial phases of my verification flow.

In Chapter 3, I reviewed the closely related work on formal verification of software specifications vs. sequential circuits. Here, I will focus on related work that can handle software with complex control flow for combinational circuits.

Fujita gave a related approach by using virtual controllers and datapaths [40]. Instead of generating hardware from the software specification, he proposed to map both the software model and hardware model to virtual controllers and datapaths, essentially synthesizing the models onto a specific structure. Therefore, the equivalence checking problem is reduced to checking the equivalence of the data transfers that are controlled by the virtual controllers. However, when the software and hardware are very different, the correspondences between the two datapaths and virtual controllers for the software and hardware are not obvious at all.

Instead of synthesis, Kroening, Clarke and Yorav [61] (see Section 3.2) give an approach using symbolic simulation with bounded model checking to both a circuit and a C program. In practice, their tool (hw-CBMC) can handle the same kinds of problems as mine. However, hw-CBMC does explicit path enumeration. Therefore, the execution time blows up with the exponential number of paths (experimental results are given later).

An alternative to path enumeration is to merge execution paths as much as possible, keeping track of the different path conditions and possible values in the symbolic expressions. Consider the following block of code:

```
if (c1) x=a;
else x=b;
if (c2) x++;
else x--;
```

Rather than analyzing each of the four execution paths separately, we could compute some sort of symbolic expression for `x` after the first `if` statement that merges the two branches, e.g., $\mathrm{ite}(c1, a, b)$, and then merge again after the second `if` statement, producing $\mathrm{ite}(c2, \mathrm{ite}(c1, a, b) + 1, \mathrm{ite}(c1, a, b) - 1)$. Merging paths converts the exponential complexity of path enumeration into logical complexity in the symbolic expressions. Early work along these lines [50, 70] suffered from BDD blow-up for non-trivial software models.

Recently, Koelbl and Pixley have proposed a more scalable approach [60]. They symbolically simulate C++ software specifications at the word level and use an acyclic circuit representation (a data flow graph) for the symbolic expressions. Starting from this data flow graph, they can do synthesis and verification. This approach greatly reduces blow-up. Furthermore, blow-up of the path conditions is reduced by a two-level representation: branching conditions in the program are abstracted as Boolean variables, and the path condition is stored as a BDD over those variables. This two-level representation allows fast approximate reasoning, but accurate computation of path conditions is expensive, requiring a combined decision procedure for — or else flattening — the two-level representation. No implementation is publicly available, but the published theory is unable to handle loop complexity like that found in the challenge problem below.

5.3 Challenge Problem: Instruction Length Decoder

Academic research on software-to-RTL verification has been stymied by the lack of good benchmark examples. Companies are reluctant to make public valuable intellectual property, and substantial engineering effort is required to create examples. I was fortunate to have a well-defined, industrial challenge problem suggested to me, which epitomizes this class of verification problem: an instruction length decoder for Intel's IA-32 (Intel Architecture, 32-bit) instruction set architecture [53]. The functionality of such a circuit (described below) is conceptually simple and easy to describe in software, although the actual code is long and has complex control flow. The RTL implementation does not resemble the high-level software. I have created a set of example software and RTL models, implementing increasingly larger and more complete decoders, and released them publicly to help other researchers.[2] (See Section 5.5.)

The IA-32 instruction set architecture (ISA) descends directly from the 16-bit Intel 8086/8088 through the latest Pentium processors. This family of processors has dominated desktop computing for over two decades and several orders of magnitude increase in processing power. Backwards software compatibility has always been important, so the ISA has grown by accretion, resulting in extremely complex instruction encodings. Instructions can range from 1 byte to over 15 bytes in length. All IA-32 instruction encodings obey the format shown as Figure 5.1. The actual length of an IA-32 instruction depends on the operating mode (protected mode, real-address mode and system management mode), the prefix bytes (if any), the opcode byte(s), the ModR/M byte (if present), and the Scale Index Base (SIB) byte (if present). For example, in protected mode, the default operand and address sizes are both 32 bits. But the operand-size override prefix (66H) and the address-size override prefix (67h) allow a program to switch to non-default operand and addressing sizes, which are 16 bits. For some instructions, with a current operand-size attribute determined by the operating mode and operand-size override prefix (if present), the size of operands can further be changed by the operand size bit (w) of the opcode. If w is 0, the operand size is 1 byte regardless of the current operand-size attribute. If w is 1, it has no effect on the current operand size. In addition, if there is a ModR/M byte field in an instruction, the 256 values of ModR/M will define different addressing forms which will affect the length of the displacement field and decide whether there is an SIB byte to follow. The addressing forms are different for different addressing modes (16-bit or 32-bit).

[2]Examples are available at `http://www.cs.ubc.ca/~ajh`.

Prefix	Opcode	ModR/M	SIB	Displacement	Immediate Data

Prefix:	0~4 Bytes	SIB:	0~1 Byte
Opcode:	1~2 Bytes	Displacement:	0~4 Bytes
ModR/M:	0~1 Byte	Immediate:	0~4 Bytes

Figure 5.1: IA-32 General Instruction Format

Because of the complex instruction encoding, a high-performance IA-32 implementation must pipeline instruction decoding. A piece of this puzzle is the instruction length decoder (ILD). (My description is based on [57].) Each cycle, the ILD is given an n-byte *parcel* of the next bytes in the instruction stream, enough additional lookahead bytes in the instruction stream to determine the end of the last instruction in the current parcel (which can extend into the next parcel), and a *wrap pointer* that indicates how far the last instruction from the previous parcel extends into this parcel. The output is two n-bit vectors `begin` and `end`, which indicate the beginning and end of each instruction in the parcel, and the wrap pointer for the next parcel.

A high-level software specification of the ILD is straightforward, if a bit convoluted. The software simply starts at the wrap pointer and scans the parcel byte-by-byte, parsing each instruction one at a time. Figure 5.2 gives pseudocode for the main loop of the software ILD (my complete C code has about 500 lines).

```
while (wrap < PARCEL_SIZE) {
  begin[wrap]=1; /* Start of instruction */

  /* Set default sizes. */
  operand_mode = INIT_OPERAND_MODE;
  address_mode = INIT_ADDRESS_MODE;

  get_next_byte();
  ret = handle_prefixes();
  /* If there were any prefixes,
     get the next byte for opcode. */
  if (ret) get_next_byte();

  if (current_byte != ESCAPE) {
    handle_one_byte_opcodes();
  } else { /* Escape to two-byte opcode */
    /* Skip over the escape code. */
    get_next_byte();
    handle_two_byte_opcodes();
  }
}
```

Figure 5.2: Software Model Main Loop Pseudocode

The loop body is basically a simple syntax-directed parser for the instruction format, with the various functions communicating via global variables, such as `wrap` pointing to the current location in the parcel or the lookahead bytes, `current_byte` being the content of that location, and the `_mode` variables remembering the current operand and address sizes. The `handle_` helper functions consist of nested `if` statements that continue the parsing for as many bytes as needed into the details of the instruction format. Note that the loop body can end with `wrap` being incremented by many different values, from 1 to the longest instruction length, depending on the context of the input parcel.

The hardware implementation is very different. For performance reasons, all decoding must be done in parallel. The logic required to decode the length of an instruction starting at a fixed location is straightforward. This logic is replicated for each possible instruction alignment in the parcel, speculatively computing the length assuming that an instruction starts there. A priority-encoding network determines which blocks of instruction-length logic are the valid ones: the length computed at the input `wrap` position is valid, and the length starting at a position j is valid iff the length starting at a position i is valid and $j = i + \mathit{length\_from}(i)$. Given the large difference between software specification and hardware implementation, verifying functional equivalence is a challenge.

5.4 Verification Algorithm

The verification algorithm I have implemented compares a high-level model, given as an annotated control-flow graph (ACFG), to a gate-level model, given in BLIF (Berkeley Logic Interchange Format [87]). The translation from a programming language like C or C++ to a CFG is standard and well-known. Starting from the CFG keeps my tool language-neutral and saves considerable implementation effort in the front-end. I also assume that all functions have been inlined, and a simple, intraprocedural dataflow analysis has been done, i.e., the analyses of reaching definitions and live variables have been done. Algorithms for these steps are available in any compiler reference, e.g., [72].

Proceeding from the annotated CFG, the verification algorithm has two main phases: a preliminary analysis of the software to unroll loops, merging paths as much as possible; and the actual formal equivalence check, where the algorithm tries to insert cutpoints during the processing of the unrolled CFG.

5.4.1 Preliminary Software Analysis

My tool handles loops in the software by unrolling. Simple loops, e.g., with constant bounds or simple dependencies, can be unrolled in the obvious manner (cf. [60]). The main instruction length decoder loop (Figure 5.2), however, has a very complex structure, so I devised a more elaborate means to perform the unrolling. Advanced compiler/synthesis optimization techniques can achieve the same result [44].

In my method, the dataflow analysis determines that `wrap` is the only loop-carried dependence (i.e., dependence between instructions from different iterations of a loop). In addition, live variable analysis shows that `wrap` is the only live variable besides the input variables, and input variables are read-only. Therefore, `wrap` is the only information that must be tracked in distinguishing different iterations of the loop body. (The paths beyond a merge point that have the same value of wrap will have equivalent future executions.) Accordingly, as the tool unrolls the loop, any two paths that (re)enter the loop with the same value of `wrap` can be merged safely. The net result is that the tool automatically unrolls the original CFG into the graph shown in Figure 5.3.[3]

[3]In [45], the authors use caching and summaries to merge paths on a particular abstraction model. The path merging uses standard approaches of data flow analysis to merge paths that have the same program states. Compared to this work, the novelty of my path merging is that my approach employs live variable analysis and merges paths that have the same values for live variables. In addition, my approach maintains full bit-accuracy.

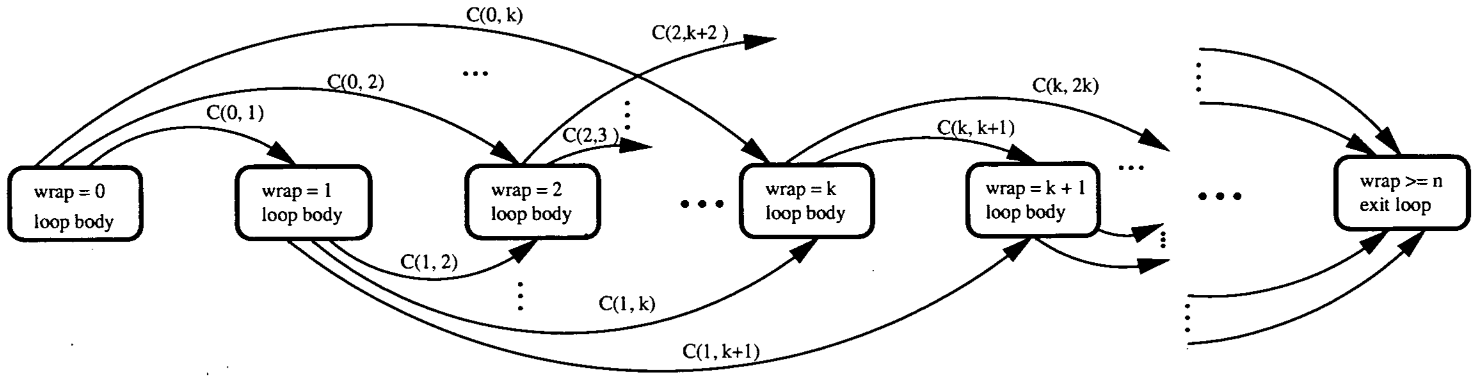


Figure 5.3: Unrolled and Merged Control Flow Graph. $C(i, j)$ denotes the logical condition such that the loop iteration with `wrap` $= i$ will continue to the loop iteration with `wrap` $= j$. For clarity, I haven't drawn the graph edges and vertices inside the loop bodies; the actual CFG, of course, does have those details.

There are two key points about the method. First is that the graph construction, and the resulting graph itself is linear in the size of the software (once the loops are unrolled). There is no exponential blow-up of path enumeration, because the algorithm explores each *edge* once, not each path. The second key point is that the tool hasn't performed a full symbolic simulation yet. Symbolic simulation would compute expressions giving the values of all the variables of the circuit, e.g., the `begin` and `end` vectors. These outputs *do* depend on the *path* taken to reach a given point in the graph. For example, whether the kth byte in a parcel is the start of an instruction depends on where the previous valid instruction ended. Thus, symbolic simulation requires computing, for each variable at each point in the unrolled graph, a logical expression that gives the correct value depending on the path taken to arrive there. These expressions are liable to blow-up.

5.4.2 Formal Equivalence Check

I now proceed to the main phase of the verification algorithm. To formally verify equivalence between the software and hardware models, the algorithm must derive some representation of the function computed at the output of each model. For both gate-level hardware or an execution path in software, symbolic simulation is the standard method to derive these representations. I will use BDDs [14] to represent the functions in symbolic simulation: they are empirically efficient, and canonicity makes proving equivalence in constant time, which facilitates finding cutpoints.

As mentioned above, the value of a variable at a given point in the program depends on the path taken through the program to reach that point. The verification algorithm, therefore, must compute the logic that indicates whether any given path is taken. For example, in Figure 5.3, the software code in loop body k will be executed (and thereby affects the values of any program variables) iff the execution starting from the initial value of `wrap` followed a path that eventually reached loop body k, i.e., there exists some sequence of values $v_0, \ldots, v_l$, where v_0 equals the initial value of wrap when the code starts, $v_l = k$, and for all i, the edge conditions $C(v_i, v_{i+1})$ are all true. (And the conditions for control flow within the loop bodies, not drawn in Figure 5.3, have to be true as well.)

Fortunately, path merging avoids enumerating the exponential number of paths. In fact, the logic that indicates whether a given basic block executes can be computed linearly in the graph size. The algorithm works as follows: Let $P(k)$ denote the *path condition* for basic block k, i.e., the logical expression that indicates what inputs will cause the software to execute basic block k. Logic $P(k)$ can be computed recursively:

$$P(k) = \bigvee_i (P(i) \wedge C(i,k))$$

i.e., for basic block k to execute, it must be true that some basic block i executed and then the branching condition $C(i,k)$ that control flowed from i to k was also true. Because the unrolled CFG is acyclic, this computation examines each edge exactly once, yielding the linear complexity.

The above linear construction is a state-of-the-art symbolic simulation approach to converting the software specification into BDDs or some other function representation. We are now ready to consider early cutpoint insertion.

The key idea of early cutpoint insertion is to look for cutpoints *during* the above computation of $P(k)$ rather than after it is completed. If the tool finds some $P(i)$ or other BDD that is equal to some point in the gate-level circuit, it cuts out the equivalent logic in the software and the hardware, and introduce a new primary input in its place. This eliminates the complexity of the logic for $P(i)$ in subsequent computations. As with gate-level cutpoints, if this process continues to the outputs (of the software and hardware) and the representations of the outputs (of the software and hardware) are equivalent, then the tool has formally verified equivalence.

For example, Figure 5.4 shows some cutpoints added to Figure 5.3. If the tool proves that the path condition for loop body 0 is the same as some logic in the gate-level circuit, It can cut out the logic and introduce a new primary input x_0 at the cut. Repeating the process introduces cuts at x_1, x_2, etc. With cutpoints, the logic for path condition $P(k)$ is simplified from

$$\bigvee_i (P(i) \wedge C(i,k))$$

to:

$$\bigvee_i (x_i \wedge C(i,k)).$$

Like most cutpoint methods, this approach is conservative: introducing cutpoints loses information and may result in being unable to prove equivalent models equivalent. In the other direction, cutpoints won't erroneously prove inequivalent models equivalent, but the algorithm may not find enough cutpoints to reduce verification complexity. Either failing is a theoretical possibility; the only way to evaluate practical usefulness is via experiments.

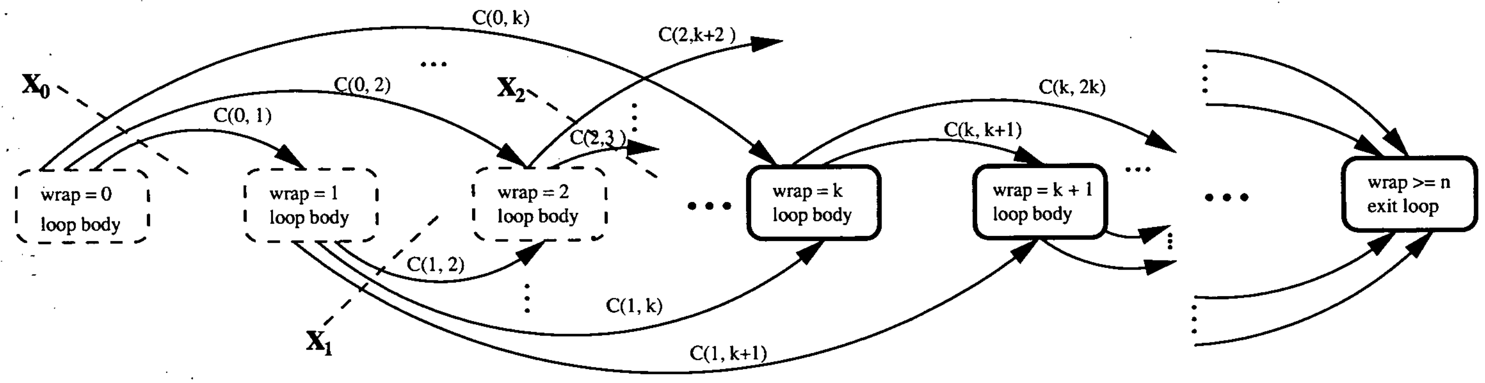


Figure 5.4: Early Cutpoint Insertion. Possible cutpoints, like x_0, etc., are checked against the hardware model and cutpoints are inserted during the analysis of the software, not after.

Instruction	Encoding	Immediate Data	Instruction Length
add(ADD)	00	0	1
add immediate(ADDI)	01	2	3
move(MOV)	10	0	1
move immediate(MOVI)	1100	2	4
store(ST)	1101	0	2
store immediate(STI)	1110	2	4

Table 5.1: Toy Example

5.5 Experimental Results

As mentioned in Section 5.3, when I started my thesis, there were no publicly available industrial examples of non-synthesizable software and corresponding hardware implementations. Accordingly, I was forced to implement such examples by myself. I have created a set of instruction length decoders of increasing size of complexity for my experiments.[4]

TOY is an example taken from [57] to describe what an instruction length decoder looks like. This toy example (Table 5.1) has only 6 instructions: 3 one-byte opcode instructions, and 3 two-byte opcode instructions. To simplify the problem further, the size of a "byte" in TOY is only 2 bits, and there are no prefixes, ModR/M, or displacement bytes. The two-byte opcode instruction format consists of an escape opcode byte as the primary opcode and a second opcode byte. Figure 5.5 and Figure 5.6 give the C function for this TOY example with the parcel size set to 4 2-bit "bytes". For this example, I have 4 sets of data with parcel sizes of 8, 16 and 32 2-bit "bytes".

[4]Examples are available at `http://www.cs.ubc.ca/~ajh`.

```
void length_decoder(
     char    *parcel,      /* 4 "bitpair" parcel */
     char    *nextparcel,  /* 4 "bitpair" next parcel (needed lookahead) */
     char    *wrapin,      /* 4 "bit" input of wrap pointer */
     char    *begin,       /* 4 "bit" output of instruction starts. */
     char    *end,         /* 4 "bit" output of instruction ends. */
     char    *wrapout      /* 4 "bit" output of wrap pointer. */
)
{
     int     wrap = 0;     /* wrap location as integer */
     int     bitpair;      /* integer value of bitpair */
     int     i;

     for (i=0; i<4; i++) {
           begin[i]=0;
           end[i]=0;
     }
     while (wrapin[wrap]==0) {
            begin[wrap]=0;
            end[wrap]=0;
            wrap++;
     }
     if (wrap>0) end[wrap-1]=1;
```

Figure 5.5: C Code for TOY Example — Part 1

```
     while (wrap<=3) {
            begin[wrap]=1;

            bitpair = 2*parcel[wrap*2] + parcel[wrap*2+1];
            switch (bitpair) {
            case 0: /* ADD */
            case 2: /* MOV */
                    wrap++;
                    end[wrap-1]=1;
                    break;
            case 1: /* ADDI */
                    wrap += 3;
                    if (wrap-1<4) end[wrap-1]=1;
                    break;
            case 3: /* Escape */
                    wrap ++;
                    if (wrap <=3)
                      bitpair = 2*parcel[wrap*2] + parcel[wrap*2+1];
                    else
                      bitpair = 2*nextparcel[0] + nextparcel[1];
                    switch (bitpair) {
                    case 0: /* MOVI */
                    case 2: /* STI */
                            wrap += 3;
                            if (wrap-1<4) end[wrap-1]=1;
                            break;
                    case 1: /* ST */
                            wrap++;
                            if (wrap-1<4) end[wrap-1]=1;
                            break;
                    case 3: /* ILLEGAL */
                            printf("Illegal instruction!\n");
                            wrap++;
                            if (wrap-1<4) end[wrap-1]=1;
                            break;
                    }
                    break;
            }
     }
     wrap -= 4;      /* Scale wrap back into range. */
     for (i=0; i<4; i++)
        wrapout[i] = (wrap==i);
}
```

Figure 5.6: C Code for TOY Example — Part 2

EX20 has 20 IA-32 instructions with lengths from 1 to 6 (8-bit) bytes. It includes three instruction formats. The first is a simple one-byte opcode instruction form without ModR/M, immediate, or displacement fields. The second is a simple two-byte opcode instruction form without ModR/M, immediate, or displacement field. The third is a one-byte opcode instruction with w bit in its opcode and with immediate data, but without ModR/M or displacement fields.

EX97 has 97 instructions (lengths from 1–8 bytes). It includes all the forms of EX20 and a new form that has a one-byte opcode instruction and ModR/M byte field.

EX251 has 251 instructions (lengths between 1–11 bytes). It includes all forms of EX97 plus a form that allows immediate data after the ModR/M byte field.

All IA-32 examples allow operand-size override and address-size override prefixes. I have created 4 different parcel sizes for each of the IA-32 examples: 8 or 12, 16, 32, and 64 bytes. (The parcel size must be larger than the longest instruction, so the smallest version of EX251 uses a 12-byte parcel.)

The high-level software is given in C code, then manually translated into my ACFG intermediate format (as defined in Chapter 3, except that the entire graph is executed in one cycle). The hardware model is given in Verilog. I use VIS [13] to translate it into BLIF and then SIS [87] with script.rugged to do optimization. As a rough indicator of complexity, I have also mapped the optimized circuits into 4-input lookup tables using Flowmap/Flowpack [26] (Table 5.2).

Example	Size
TOY-8	138
TOY-16	331
TOY-32	723
EX20-8	467
EX20-16	912
EX20-32	2251
EX20-64	9012

Example	Size
EX97-8	1637
EX97-16	3255
EX97-32	6448
EX97-64	17540
EX251-12	6199
EX251-16	8312
EX251-32	16770
EX251-64	131002

Table 5.2: Circuit Sizes of Examples. Size is the number of 4-LUTs for the synthesized hardware model.

All experiments were run on a 2.6Ghz Pentium 4 with 4GB of RAM. I set the runtime limit to 2 hours and the memory usage limit to 2GB. Memory usage is the peak as reported by `top`. For BDD experiments, memory usage varies depending on machine memory size, because the CUDD package [89] aggressively pre-allocates memory. However, runs with different memory sizes produced the same comparative results. All times are for the full verification. The verification tool proves equivalence on all examples, showing that the abstraction introduced by the cutpoints was not too conservative.

5.5.1 Avoiding Path Enumeration

As mentioned earlier, we can always enumerate execution paths in the software specification, and prove the equivalence for each path. This is done by proving under the same path condition and same inputs that the software and hardware models have equivalent outputs.

The first experiment is to compare path enumeration with the linear BDD construction from Section 5.4.2. This measures the effect of path merging in eliminating the exponential path enumeration at the cost of a possible expression-size blow-up. Table 5.3 gives the results. The execution time of path enumeration blows up all but small instances. The linear-time BDD building runs much faster by avoiding explicit exploration of the paths.

	Path Enumeration		Linear BDD	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	2.25	57	0.02	56
TOY-16	time out		5.35	56
TOY-32	time out			mem out
EX20-8	241.24	28	0.28	61
EX20-16	time out		89.01	1746
EX20-32	time out			mem out
EX20-64	time out			mem out
EX97-8	4229.44	183	1.46	92
EX97-16	time out		1187.72	1800
EX97-32	time out			mem out
EX97-64	time out			mem out
EX251-12	time out		309.18	1843
EX251-16	time out			mem out
EX251-32	time out			mem out
EX251-64	time out			mem out

Table 5.3: Path Enumeration vs. Linear BDD. Run time limit is set to 2 hour and memory limit is set to 2 GB.

5.5.2 Effect of Early Cutpoints

Next, I examine the effect of inserting cutpoints early. Table 5.4 compares my verification tool using the linear BDD construction without and with early cutpoint insertion. We see that the early cutpoint method vastly reduces both memory usage and run time.

	Linear BDD		Early Cutpoint	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	0.02	56	0.01	56
TOY-16	5.35	56	0.02	56
TOY-32		mem out	0.06	56
EX20-8	0.28	61	0.11	58
EX20-16	89.01	1746	0.24	60
EX20-32		mem out	0.53	64
EX20-64		mem out	1.35	72
EX97-8	1.46	92	0.51	64
EX97-16	1187.72	1800	1.10	73
EX97-32		mem out	2.35	95
EX97-64		mem out	5.41	136
EX251-12	309.18	1843	0.64	66
EX251-16		mem out	1.09	71
EX251-32		mem out	7.45	170
EX251-64		mem out	16.81	327

Table 5.4: Linear BDD vs. Early Cutpoint. Run time limit is set to 2 hour and memory limit is set to 2 GB.

5.5.3 Comparison with Other Tools

I know of only one freely available software-to-RTL verification tool that can handle the software complexity of my challenge problem: hw-CBMC.[5] As mentioned earlier, hw-CBMC does path enumeration [23], so it can handle only the smaller instances of my TOY example. Table 5.5 shows results compared to my early cutpoint method. This is not a fair comparison, since hw-CBMC parses arbitrary ANSI-C, whereas my tool starts from a CFG and exploits some assumptions about program structure. Nevertheless, the benefit of early cutpoint insertion and not enumerating paths is clear.

	hw-CBMC		Early Cutpoint	
Example	Time(s)	Mem(MB)	Time(s)	Mem(MB)
TOY-8	6.84	38	0.01	56
TOY-16	502.59	522	0.02	56
TOY-32	time out		0.06	56

Table 5.5: hw-CBMC vs. Early Cutpoints. Run time limit is set to 2 hour and memory limit is set to 2 GB.

[5]Version 1.6 from http://www.cs.cmu.edu/~modelcheck/cbmc.

Chapter 6

Conclusion and Future Work

6.1 Contributions

Addressing the formal equivalence checking of software specification vs. hardware implementations, I have presented two major contributions to increase the capability of current verification techniques: partitioned model checking and cutpoints for equivalence checking with software models.

My first contribution, the partitioned model checking approach, uses *Annotated Control Flow Graphs* (ACFG) to represent cycle-accurate software specifications for sequential circuits. I have introduced a novel, partitioned model checking algorithm for verifying RTL hardware against cycle-accurate software. In my experimental results, ACFG model checking runs 1.3x to 2.0x faster than standard model checking.

My second contribution, cutpoints for equivalence checking with software models, was inspired by the efficiency of combinational equivalence checking using cutpoints. I give a definition of cutpoints for software and show vast improvements on the formal equivalence checking of embedded software. Furthermore, I introduce early cutpoint insertion into the formal equivalence verification of software specifications vs. hardware implementations. I have run experiments for an industry-suggested challenge problem. Experimental results show that early cutpoint insertion has orders of magnitude improvements in both runtime and memory usage, enabling verification of all of my 15 instances of the challenge problem, whereas previous methods solve only 7 of them.

6.2 Future Work

ACFG partitioned model checking and early cutpoint insertion are first steps towards the general problem of verifying RTL hardware against higher-level, but cycle-accurate, specifications. This section considers several future research directions.

More Hints from Program Analysis and Compiler Optimization

Since a software specification gives a high-level abstraction of a hardware implementation, good understandings of the software can help us to find the correspondences between the software and the high-quality hardware implementation. For example, in my examples, the big difference of the software specification and the hardware implementation is the amount of parallelism. The hardware implementation exposes as much parallelism as possible for performance reasons. The software, on the contrary, gives serial computations for easy coding.

I believe that with the help of software analysis and compiler optimizations, which have been studied extensively for a long time, we can have more general approaches to handle complicated control structures and extract more parallelism from the software specification. Therefore, we can catch more similarities between the software specification and the hardware implementation. As a result, more cutpoints can be used to reduce the complexity of verification.

Heuristics for Finding Candidate Cutpoints

My current approach relies on structural information for candidate cutpoints. More general approaches are needed to identify the candidate cutpoints automatically.

There are several possible solutions to find candidate cutpoints. Like hardware combinational equivalence checking, we can run random simulation on both software specifications and hardware implementations and use signature-based approaches to find candidate cutpoints. However, random simulation on software cannot give good coverage. For example, at each branch, random simulation must make a single choice. Therefore, random simulation can not cover some piece of code in the software. How to combine fast random simulations with candidate cutpoints identification is still very challenging and needs research.

False Inequivalence Handling

Just like the work for embedded software, I need to handle false inequivalences for the early cutpoint approach in equivalence checking of software specifications vs. hardware implementations. But, in my test case, I don't have false inequivalences, so I didn't deal with this problem.

The obvious approach is to apply standard techniques from combinational equivalence checking to handle false inequivalences by re-introducing constraints on the cutpoints. Future work would investigate whether there are optimizations specifically based on the software specification.

Integration with Other Advances

Partitioned model checking and early cutpoint insertion improve two aspects of the overall equivalence verification flow. Important future work will be to combine them with the best ideas for other parts of this flow, e.g., preliminary textual pruning [81], a full-fledged software front-end [23], powerful software analyses and optimizations [44], and more general and efficient symbolic representations [60]. Industrial-strength high-level-to-RTL equivalence verification will require many advances; ACFG-based partitioned model checking and early cutpoint insertion are two of them.

Bibliography

[1] Samir Agrawal and Rajesh K. Gupta. Data-flow assisted behavioral partitioning for embedded systems. In *DAC '97: Proceedings of the 34th Design Automation Conference*, pages 709–712. ACM Press, 1997.

[2] S.B. Akers. Binary decision diagrams. *IEEE Transation on Computers*, 27(6):509–516, June 1978.

[3] S. Balakrishnan and S. Tahar. On the formal verification of embedded systems using multiway decision graphs. Technical Report TR-402, Concordia University, Montreal, Canada, 1997.

[4] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[5] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN*, pages 113–130, 2000.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer-Aided Verification: 13th International Conference*, number 2102 in Lecture Notes in Computer Science, pages 260–264. Springer, 2001.

[7] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design: First International Conference*, volume 1166 of *Lecture Notes in Computer*

Science, pages 187–201. Springer, 1996. Currently, software is available at `http://chicory.stanford.edu/SVC`.

[8] Bob Bentley. High level validation of next generation microprocessors. In *International Workshop on High-Level Design, Validation, and Test*, pages 31–35. IEEE, 2002.

[9] C. Leonard Berman and Louise H. Trevillyan. Functional comparison of logic designs for VLSI circuits. In *International Conference on Computer-Aided Design*, pages 456–459. IEEE, 1989.

[10] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of TACAS 1999*, pages 193–207, 1999.

[11] Claudia Blank, Hans Eveking, Jens Levihn, and Gerd Ritter. Symbolic simulation techniques — state-of-the-art and applications. In *International Workshop on High-Level Design, Validation, and Test*, pages 45–50. IEEE, 2001.

[12] Daniel Brand. Verification of large synthesized designs. In *International Conference on Computer-Aided Design*, pages 534–537. IEEE/ACM, 1993.

[13] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In *Computer-Aided Verification: 8th International Conference*, pages 428–432. Springer, 1996. Lecture Notes in Computer Science Number 1102.

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

[15] Randal E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, April 1991.

[16] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification: 14th International Conference*, pages 78–92. Springer, 2002. Lecture Notes in Computer Science Vol. 2404.

[17] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI '91: International Conference on Very Large Scale Integration*, Edinburgh, Great Britain, 1991. IFIP TC 10/WG 10.5.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990. An extended version of this paper appeared in *Information and Computation*, Vol. 98, No. 2, June 1992.

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.

[20] Kwang-Ting Cheng and Vishwani D. Agrawal. *Unified Methods for VLSI Simulation and Test Generation.* Kluwer Academic Publishers, 1989.

[21] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification: Eighth International Conference*, pages 419–422. Springer-Verlag, July 1996. Lecture Notes in Computer Science Number 1102.

[22] Edmund Clarke and Daniel Kroening. Behavior consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, May 2003.

[23] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Asia South-Pacific Design Automation Conference*, pages 308–311. IEEE, 2003.

[24] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.

[25] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[26] Jason Cong and Yuzheng Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, January 1994.

[27] S. Cook. The complexity of theorem-proving procedures. In *The 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[28] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardike. Benefits of bounded model checking at an industrial setting. In *Proceedings of CAV 2001*, pages 436–453, 2001.

[29] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, March 2006.

[30] David W. Currie, Alan J. Hu, Sreeranga Rajan, and Masahiro Fujita. Automatic formal verification of DSP software. In *37th Design Automation Conference*, pages 130–135. ACM/IEEE, 2000.

[31] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 16, pages 995–1072. MIT Press, 1994.

[32] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. In *Symposium on Principles of Programming Languages*, pages 127–140. ACM, 1983.

[33] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[34] Xiushan Feng. Automatic formal verification for scheduled VLIW code. Master's thesis, University of British Columbia, August 2002.

[35] Xiushan Feng and Alan J. Hu. Automatic formal verification for scheduled VLIW code. In *Joint Conference on Languages, Compilers, and Tools for Embedded Systems, and Software and Compilers for Embedded Systems*, pages 85–92. ACM SIGPLAN, 2002.

[36] Xiushan Feng and Alan J. Hu. Cutpoints for formal equivalence verification of embedded software. In *5th International Conference on Embedded Software*, pages 307–316. ACM, 2005.

[37] Xiushan Feng and Alan J. Hu. Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification. In *Proceedings of the 43rd Design Automation Conference (DAC)*, pages 1063–1068. ACM SIGDA, 2006.

[38] Xiushan Feng, Alan J. Hu, and Jin Yang. Partitioned model checking from software specifications. In *Asia South Pacific Design Automation Conference*, pages 583–587. IEEE Press, 2005.

[39] Hiroshige Fujii, Goichi Ootomo, and Chikahiro Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 38–41. IEEE, 1993.

[40] Masahiro Fujita. Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(4):610–626, 2005.

[41] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. on Soft. Eng.*, 19(3):214–230, 1993.

[42] Mike Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[43] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[44] Sumit Gupta, Timothy Kam, Michael Kishinevsky, Shai Rotem, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Proceedings of the 39th on Design Automation Conference*, pages 898–903, 2002.

[45] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system-specific, static analyses. In *Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[46] Kiyoharu Hamaguchi, Hidekazu Urushihara, and Toshinobu Kashiwabara. Symbolic checking of signal-transition consistency for verifying high-level designs. In *Formal Methods in Computer-Aided Design: Third International*

Conference, pages 455–469. Springer, 2000. Lecture Notes in Computer Science Vol. 1954.

[47] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Conference on Principles of Programming Languages*, pages 58–70. ACM SIGPLAN-SIGACT, 2002.

[48] Ramin Hojati, Sriram C. Krishnan, and Robert K. Brayton. Early quantification and partitioned transition relations. In *International Conference on Computer Design*, pages 12–19, 1996.

[49] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Computer-Aided Verification: Fifth International Conference.* Springer-Verlag, 1993. Lecture Notes in Computer Science Number 697.

[50] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop.* Springer-Verlag, July 1992. Published in 1993 as Lecture Notes in Computer Science Number 663.

[51] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282. ACM/IEEE, 1994.

[52] Michael Huth and Mark Ryan. *Logic in Computer Science.* Cambridge University Press, 2006.

[53] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual*, 2004. Four volumes. Intel Order Numbers 253665–253668.

[54] Himanshu Jain, Daniel Kroening, and Edmund M. Clarke. Verification of SpecC using predicate abstraction. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 7–16. IEEE, 2004.

[55] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 445–454, Washington, October 1997. IEEE.

[56] Jawahar Jain, Amit Narayan, Masahiro Fujita, and Alberto Sangiovanni-Vincentelli. Formal verification of combinational circuits. In *International Conference on VLSI Design*, pages 218–225, 1997.

[57] Robert B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors.* PhD thesis, Stanford University, 1999.

[58] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[59] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[60] Alfred Koelbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simuluation. *International Journal of Parallel Programming*, 33(6):645–666, December 2005.

[61] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, 2003.

[62] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *34th Design Automation Conference*, pages 263–268. ACM/IEEE, 1997.

[63] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, 1988.

[64] C.Y. Lee. Representations of the switching circuits by binary decision diagrams. *Bell Systems Technical Journal*, July 1959.

[65] Zohar Manna and Amir Pnueli. Specification and verification of concurrent programs by ∀-automata. In *Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1987.

[66] João P. Marques Silva and Karem A. Sakallah. GRASP — a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227. IEEE/ACM, 1996.

[67] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C-based hardware descriptions by using symbolic simulation and program slicer. In *Proceedings of IWLS 2003*, 2003.

[68] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie-Mellon University, May 1992. Published as CMU Tech Report CMU-CS-92-131.

[69] Christoph Meinel and Anna Slobodová. Speeding up variable reordering of OBDDs. In *International Conference on Computer Design*, pages 338–343. IEEE, October 1997.

[70] Shin-ichi Minato. Generation of BDDs from hardware algorithm descriptions. In *International Conference on Computer-Aided Design*, pages 644–649. IEEE/ACM, 1996.

[71] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference*, pages 530–535. ACM/IEEE, 2001.

[72] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[73] Amit Narayan, Jawahar Jain, Masahiro Fujita, and Alberto Sangiovanni-Vincentelli. Partitioned-ROBDDs — a compact, canonical and efficiently manipulable representation for Boolean functions. In *International Conference on Computer-Aided Design*. IEEE/ACM, 1996.

[74] George C. Necula. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation*, pages 83–94. ACM SIGPLAN, 2000.

[75] Rob Oshana. Optimization techniques for high-performance DSPs. *Embedded Systems Programming*, March 1999. We accessed the on-line article at `http://www.embedded.com/1999/9903/9903osha.htm`.

[76] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *International Conference on Automated Deduction*, pages 748–752, 1992. Lecture Notes in Artificial Intelligence Number 607.

[77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–55, 1977.

[78] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981. Lecture Notes in Computer Science Number 137.

[79] Rajeev K. Ranjan, Wilsin Gosti, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: Challenges and solutions. In *International Conference on Computer Design*, pages 344–351. IEEE, October 1997.

[80] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.

[81] Hiroshi Saito, Takaya Ogawa, Thanyapat Sakunkonchak, Masahiro Fujita, and Takashi Nanya. An equivalence checking methodology for hardware oriented C-based specifications. In *International High-Level Design, Validation, and Test Workshop*, pages 139–144. IEEE, 2002.

[82] Hanan Samet. *Automatically proving the correctness of translations involving optimized code.* PhD thesis, Stanford University, 1975.

[83] Tom Schubert. High level formal verification of next-generation microprocessors. In *40th Design Automation Conference*, pages 1–6. ACM/IEEE, 2003.

[84] Roberto Sebastiani, Eli Singerman, Stefano Tonetta, and Moshe Y. Vardi. GSTE is partitioned model checking. In *Computer-Aided Verification: 16th International Conference*, pages 229 – 241. Springer-Verlag, 2004. Lecture Notes in Computer Science Number 3114.

[85] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, 1995.

[86] Luc Séméria, Andrew Seawright, Renu Mehra, Daniel Ng, Arjuna Ekanayake, and Barry Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *39th Design Automation Conference*, pages 123–128. ACM/IEEE, 2002.

[87] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, University of California Berkeley, May 1992.

[88] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer-Aided Verification: 12th International Conference*, pages 480–494. Springer, 2000. Lecture Notes in Computer Science Vol. 1855.

[89] Fabio Somenzi. CUDD: CU decision diagram package. Available from `ftp://vlsi.colorado.edu/pub/`.

[90] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *Computer-Aided Verification: 14th International Conference*, pages 500–504. Springer, 2002. Lecture Notes in Computer Science Vol. 2404.

[91] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. of Mathematics*, 5:285–309, 1955.

[92] Texas Instruments. *TMS320C67x DSP Library*. Version 1.00, February 17, 2003. Part Number SPRC121. `http://focus.ti.com/docs/toolsw/folders/print/sprc121.html`.

[93] O. Thiry and L. Claesen. A formal verification technique for embedded software. In *IEEE International Conference on Computer Design*, pages 352–357, New York, USA, 1996. IEEE Computer Society Press.

[94] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[95] Jin Yang and Amit Goel. GSTE through a case study. In *International Conference on Computer-Aided Design*, pages 534–541. IEEE/ACM, 2002.

[96] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. In *International Conference on Computer Design*, pages 360–365. IEEE, 2001.

[97] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation — abstraction in action. In *Formal Methods in Computer-Aided Design: Fourth International Conference*, pages 70–87. Springer, 2002. Lecture Notes in Computer Science Number 2517.

[98] Joon-Seo Yim, Yoon-Ho Hwang, Chang-Jae Park, Hoon Choi, Woo-Seung Yang, Hun-Seung Oh, In-Cheol Park, and Chong-Min kyung. A C-based RTL design verification methodology for complex microprocessor. In *Proceedings of DAC 1997*, 1997.

[99] Hantao Zhang. SATO: An efficient propositional prover. In *14th Conference on Automated Deduction*, pages 272–275. Springer, 1997. Lecture Notes in Artificial Intelligence Vol. 1249.

[100] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation for optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

Appendix A

ACFG Specification of SRT Divider

```
// ACFG specification for a SRT unsigned divider for Chapter 3
#define N = 4 // 2N-bit dividend, N-bit divisor
Vertex: 2   // number of vertices
Edge: 4     // number of edges
Var: 9      // number of variables
 // input variables
 input_enable: reg[0..0];
 dividend: reg[2*N-1..0];
 divisor: reg[N-1..0];

 // output variables
 quotient: reg[N-1..0];
 remainder: reg[N-1..0];

 // internal variables
 P: reg[2*N..0];
 qp: reg[N-1..0];
 qn: reg[N-1..0];

 // define inputs and outputs
 Input_pin[3]: dividend,divisor,input_enable;
 Output_pin[2]: remainder, quotient;

 Initial: Vertex_0; // entry vertex

// Define ACFG edges

Edge_0: Vertex_0 -> Vertex_1
  input_enable == 1             // antecedent
   ==>
   qp := 0 && qn := 0           // assignments
 --> P[2*N-1..0]:= dividend && P[2*N] := 0
 --> remainder := P[2*N-1..N]&& quotient := qp - qn;

//  -1 case
Edge_1: Vertex_1 -> Vertex_1
  P < 7*2^(2*N - 2) && P > 2^(2*N) && input_enable == 0
   ==>
  P[2*N-1..N-1] := P[2*N-1.. N-1] + divisor
  -->qp := qp + qp && qn := qn + qn && P:=P+P
  -->qn := qn xor 1
  -->P[2*N] ==0
   && remainder := P[2*N-1..N] && quotient := qp - qn
    ||
   P[2*N] ==1
   && remainder := P[2*N-1..N] + divisor && quotient := qp - qn - 1;

// 0 case
Edge_2: Vertex_1 -> Vertex_1
   (P < 2^(2*N -2) || P >= 7*2^(2*N - 2)) && input_enable == 0
   ==>
   qp := qp + qp && qn := qn + qn && P:=P+P
   -->P[2*N] ==0
    && remainder := P[2*N-1..N] && quotient := qp - qn
     ||
    P[2*N] ==1
    && remainder := P[2*N-1..N] + divisor && quotient := qp - qn - 1;

// 1 case
Edge_3: Vertex_1 -> Vertex_1
  P >= 2^(2*N -2) && P < 2^(2*N) && input_enable == 0
   ==>
    P[2*N-1..N-1] := P[2*N-1.. N-1] -  divisor
   -->qp := qp + qp && qn := qn + qn && P:=P+P
   -->qp := qp xor 1
   -->P[2*N] ==0
    && remainder := P[2*N-1..N] && quotient := qp - qn
     ||
    P[2*N] ==1
    && remainder := P[2*N-1..N] + divisor && quotient := qp - qn - 1;
end
```

Appendix B

SRT Divider Gate-Level Circuit

```
# Example circuit for Chapter 3
# Circuit is written in modified ISCAS'89 format
# Inputs are dividend_2N-1 downto dividend_0
# and divisor_N-1 downto divisor_0

# Results pop out N+1 clock cycles later in
# quotient_N-1 downto quotient_0
# and remainder_N-1 downto remainder_0

# divisor must be normalized (leading bit 1)
# not checking for overflow

# Be sure to initialize the input_enable latch to 1.

# P is 2N bit partial remainder (P_2N is extra sign bit)
# D is N bit divisor
# QP is N bit positive quotient bits
# QN is N bit negative quotient bits
# In this circuit example, N = 4. It is easy to get circuits
# for different N from the pattern of this circuit.

# PREAMBLE

# Declare Inputs and Outputs
INPUT(dividend_7)
INPUT(dividend_6)
INPUT(dividend_5)
INPUT(dividend_4)
INPUT(dividend_3)
INPUT(dividend_2)
INPUT(dividend_1)
INPUT(dividend_0)
INPUT(divisor_3)
INPUT(divisor_2)
INPUT(divisor_1)
INPUT(divisor_0)
INPUT(input_enable)
OUTPUT(quotient_3)
OUTPUT(quotient_2)
OUTPUT(quotient_1)
OUTPUT(quotient_0)
OUTPUT(remainder_3)
OUTPUT(remainder_2)
OUTPUT(remainder_1)
OUTPUT(remainder_0)
```

```
# Load logic for registers
input_disable = NOT(input_enable)
LOW = AND(input_enable,input_disable)
# Load enabled values
sel_load_P_8 = AND(input_enable,input_disable)
sel_load_P_7 = AND(input_enable,dividend_7)
sel_load_P_6 = AND(input_enable,dividend_6)
sel_load_P_5 = AND(input_enable,dividend_5)
sel_load_P_4 = AND(input_enable,dividend_4)
sel_load_P_3 = AND(input_enable,dividend_3)
sel_load_P_2 = AND(input_enable,dividend_2)
sel_load_P_1 = AND(input_enable,dividend_1)
sel_load_P_0 = AND(input_enable,dividend_0)
sel_load_D_3 = AND(input_enable,divisor_3)
sel_load_D_2 = AND(input_enable,divisor_2)
sel_load_D_1 = AND(input_enable,divisor_1)
sel_load_D_0 = AND(input_enable,divisor_0)
sel_load_QP_3 = AND(input_enable,input_disable)
sel_load_QP_2 = AND(input_enable,input_disable)
sel_load_QP_1 = AND(input_enable,input_disable)
sel_load_QP_0 = AND(input_enable,input_disable)
sel_load_QN_3 = AND(input_enable,input_disable)
sel_load_QN_2 = AND(input_enable,input_disable)
sel_load_QN_1 = AND(input_enable,input_disable)
sel_load_QN_0 = AND(input_enable,input_disable)
sel_load_DIVIDEND_7 = AND(input_enable,dividend_7)
sel_load_DIVIDEND_6 = AND(input_enable,dividend_6)
```

```
sel_load_DIVIDEND_5 = AND(input_enable,dividend_5)
sel_load_DIVIDEND_4 = AND(input_enable,dividend_4)
sel_load_DIVIDEND_3 = AND(input_enable,dividend_3)
sel_load_DIVIDEND_2 = AND(input_enable,dividend_2)
sel_load_DIVIDEND_1 = AND(input_enable,dividend_1)
sel_load_DIVIDEND_0 = AND(input_enable,dividend_0)
# Run-the-divider values
sel_run_P_8 = AND(input_disable,stage0_new_4)
sel_run_P_7 = AND(input_disable,stage0_new_3)
sel_run_P_6 = AND(input_disable,stage0_new_2)
sel_run_P_5 = AND(input_disable,stage0_new_1)
sel_run_P_4 = AND(input_disable,stage0_new_0)
sel_run_P_3 = AND(input_disable,stage0_P_2)
sel_run_P_2 = AND(input_disable,stage0_P_1)
sel_run_P_1 = AND(input_disable,stage0_P_0)
sel_run_P_0 = AND(input_disable,input_enable)
sel_run_D_3 = AND(input_disable,stage0_D_3)
sel_run_D_2 = AND(input_disable,stage0_D_2)
sel_run_D_1 = AND(input_disable,stage0_D_1)
sel_run_D_0 = AND(input_disable,stage0_D_0)
sel_run_QP_3 = AND(input_disable,stage0_QP_2)
sel_run_QP_2 = AND(input_disable,stage0_QP_1)
sel_run_QP_1 = AND(input_disable,stage0_QP_0)
sel_run_QP_0 = AND(input_disable,stage0_qb1)
sel_run_QN_3 = AND(input_disable,stage0_QN_2)
sel_run_QN_2 = AND(input_disable,stage0_QN_1)
sel_run_QN_1 = AND(input_disable,stage0_QN_0)
sel_run_QN_0 = AND(input_disable,stage0_qbn1)
```

```
sel_run_DIVIDEND_7 = AND(input_disable,stage0_DIVIDEND_7)
sel_run_DIVIDEND_6 = AND(input_disable,stage0_DIVIDEND_6)
sel_run_DIVIDEND_5 = AND(input_disable,stage0_DIVIDEND_5)
sel_run_DIVIDEND_4 = AND(input_disable,stage0_DIVIDEND_4)
sel_run_DIVIDEND_3 = AND(input_disable,stage0_DIVIDEND_3)
sel_run_DIVIDEND_2 = AND(input_disable,stage0_DIVIDEND_2)
sel_run_DIVIDEND_1 = AND(input_disable,stage0_DIVIDEND_1)
sel_run_DIVIDEND_0 = AND(input_disable,stage0_DIVIDEND_0)
mux_P_8 = OR(sel_load_P_8,sel_run_P_8)
mux_P_7 = OR(sel_load_P_7,sel_run_P_7)
mux_P_6 = OR(sel_load_P_6,sel_run_P_6)
mux_P_5 = OR(sel_load_P_5,sel_run_P_5)
mux_P_4 = OR(sel_load_P_4,sel_run_P_4)
mux_P_3 = OR(sel_load_P_3,sel_run_P_3)
mux_P_2 = OR(sel_load_P_2,sel_run_P_2)
mux_P_1 = OR(sel_load_P_1,sel_run_P_1)
mux_P_0 = OR(sel_load_P_0,sel_run_P_0)
mux_D_3 = OR(sel_load_D_3,sel_run_D_3)
mux_D_2 = OR(sel_load_D_2,sel_run_D_2)
mux_D_1 = OR(sel_load_D_1,sel_run_D_1)
mux_D_0 = OR(sel_load_D_0,sel_run_D_0)
mux_QP_3 = OR(sel_load_QP_3,sel_run_QP_3)
mux_QP_2 = OR(sel_load_QP_2,sel_run_QP_2)
mux_QP_1 = OR(sel_load_QP_1,sel_run_QP_1)
mux_QP_0 = OR(sel_load_QP_0,sel_run_QP_0)
mux_QN_3 = OR(sel_load_QN_3,sel_run_QN_3)
mux_QN_2 = OR(sel_load_QN_2,sel_run_QN_2)
mux_QN_1 = OR(sel_load_QN_1,sel_run_QN_1)
```

```
mux_QN_0 = OR(sel_load_QN_0,sel_run_QN_0)
mux_DIVIDEND_7 = OR(sel_load_DIVIDEND_7,sel_run_DIVIDEND_7)
mux_DIVIDEND_6 = OR(sel_load_DIVIDEND_6,sel_run_DIVIDEND_6)
mux_DIVIDEND_5 = OR(sel_load_DIVIDEND_5,sel_run_DIVIDEND_5)
mux_DIVIDEND_4 = OR(sel_load_DIVIDEND_4,sel_run_DIVIDEND_4)
mux_DIVIDEND_3 = OR(sel_load_DIVIDEND_3,sel_run_DIVIDEND_3)
mux_DIVIDEND_2 = OR(sel_load_DIVIDEND_2,sel_run_DIVIDEND_2)
mux_DIVIDEND_1 = OR(sel_load_DIVIDEND_1,sel_run_DIVIDEND_1)
mux_DIVIDEND_0 = OR(sel_load_DIVIDEND_0,sel_run_DIVIDEND_0)
stage0_P_8 = DFF(mux_P_8)
stage0_P_7 = DFF(mux_P_7)
stage0_P_6 = DFF(mux_P_6)
stage0_P_5 = DFF(mux_P_5)
stage0_P_4 = DFF(mux_P_4)
stage0_P_3 = DFF(mux_P_3)
stage0_P_2 = DFF(mux_P_2)
stage0_P_1 = DFF(mux_P_1)
stage0_P_0 = DFF(mux_P_0)
stage0_D_3 = DFF(mux_D_3)
stage0_D_2 = DFF(mux_D_2)
stage0_D_1 = DFF(mux_D_1)
stage0_D_0 = DFF(mux_D_0)
stage0_QP_3 = DFF(mux_QP_3)
stage0_QP_2 = DFF(mux_QP_2)
stage0_QP_1 = DFF(mux_QP_1)
stage0_QP_0 = DFF(mux_QP_0)
stage0_QN_3 = DFF(mux_QN_3)
stage0_QN_2 = DFF(mux_QN_2)
```

```
stage0_QN_1 = DFF(mux_QN_1)
stage0_QN_0 = DFF(mux_QN_0)
# save dividend to simplify verification
stage0_DIVIDEND_7 = DFF(mux_DIVIDEND_7)
stage0_DIVIDEND_6 = DFF(mux_DIVIDEND_6)
stage0_DIVIDEND_5 = DFF(mux_DIVIDEND_5)
stage0_DIVIDEND_4 = DFF(mux_DIVIDEND_4)
stage0_DIVIDEND_3 = DFF(mux_DIVIDEND_3)
stage0_DIVIDEND_2 = DFF(mux_DIVIDEND_2)
stage0_DIVIDEND_1 = DFF(mux_DIVIDEND_1)
stage0_DIVIDEND_0 = DFF(mux_DIVIDEND_0)

# NEXT STATE LOGIC

# Compute quotient bit in redundant form
stage0_eq1 = XNOR(stage0_P_8,stage0_P_7)
stage0_eq2 = XNOR(stage0_P_7,stage0_P_6)
stage0_qb0 = AND(stage0_eq1,stage0_eq2)
stage0_qb1 = NOR(stage0_qb0,stage0_P_8)
stage0_qbn1 = NOR(stage0_qb0,stage0_qb1)
# Compute off = +d, -d, or 0 in preparation for addition
stage0_selposd_4 = AND(stage0_qbn1,LOW)
stage0_selposd_3 = AND(stage0_qbn1,stage0_D_3)
stage0_selposd_2 = AND(stage0_qbn1,stage0_D_2)
stage0_selposd_1 = AND(stage0_qbn1,stage0_D_1)
stage0_selposd_0 = AND(stage0_qbn1,stage0_D_0)
stage0_negd_4 = NOT(LOW)
stage0_negd_3 = NOT(stage0_D_3)
```

```
stage0_negd_2 = NOT(stage0_D_2)
stage0_negd_1 = NOT(stage0_D_1)
stage0_negd_0 = NOT(stage0_D_0)
stage0_selnegd_4 = AND(stage0_qb1,stage0_negd_4)
stage0_selnegd_3 = AND(stage0_qb1,stage0_negd_3)
stage0_selnegd_2 = AND(stage0_qb1,stage0_negd_2)
stage0_selnegd_1 = AND(stage0_qb1,stage0_negd_1)
stage0_selnegd_0 = AND(stage0_qb1,stage0_negd_0)
stage0_off_4 = OR(stage0_selposd_4,stage0_selnegd_4)
stage0_off_3 = OR(stage0_selposd_3,stage0_selnegd_3)
stage0_off_2 = OR(stage0_selposd_2,stage0_selnegd_2)
stage0_off_1 = OR(stage0_selposd_1,stage0_selnegd_1)
stage0_off_0 = OR(stage0_selposd_0,stage0_selnegd_0)
# Adds the quotient bit times divisor to partial remainder
stage0_new_0 = SUM(stage0_P_3,stage0_off_0,stage0_qb1)
stage0_co_0 = CARRY(stage0_P_3,stage0_off_0,stage0_qb1)
stage0_new_1 = SUM(stage0_P_4,stage0_off_1,stage0_co_0)
stage0_co_1 = CARRY(stage0_P_4,stage0_off_1,stage0_co_0)
stage0_new_2 = SUM(stage0_P_5,stage0_off_2,stage0_co_1)
stage0_co_2 = CARRY(stage0_P_5,stage0_off_2,stage0_co_1)
stage0_new_3 = SUM(stage0_P_6,stage0_off_3,stage0_co_2)
stage0_co_3 = CARRY(stage0_P_6,stage0_off_3,stage0_co_2)
stage0_new_4 = SUM(stage0_P_7,stage0_off_4,stage0_co_3)
stage0_co_4 = CARRY(stage0_P_7,stage0_off_4,stage0_co_3)

# OUTPUT LOGIC

# Subtract QN from QP to get non-redundant quotient
```

```
negQN_0 = NOT(stage0_QN_0)
negQN_1 = NOT(stage0_QN_1)
negQN_2 = NOT(stage0_QN_2)
negQN_3 = NOT(stage0_QN_3)
pos_r = NOT(stage0_P_8) # adjust for neg remainder
quotient_0 = SUM(stage0_QP_0,negQN_0,pos_r)
qnr_co_0 = CARRY(stage0_QP_0,negQN_0,pos_r)
quotient_1 = SUM(stage0_QP_1,negQN_1,qnr_co_0)
qnr_co_1 = CARRY(stage0_QP_1,negQN_1,qnr_co_0)
quotient_2 = SUM(stage0_QP_2,negQN_2,qnr_co_1)
qnr_co_2 = CARRY(stage0_QP_2,negQN_2,qnr_co_1)
quotient_3 = SUM(stage0_QP_3,negQN_3,qnr_co_2)
qnr_co_3 = CARRY(stage0_QP_3,negQN_3,qnr_co_2)
# Add divisor back into remainder if remainder is negative
neg_r = NOT(pos_r)
masked_d_0 = AND(stage0_D_0, neg_r)
masked_d_1 = AND(stage0_D_1, neg_r)
masked_d_2 = AND(stage0_D_2, neg_r)
masked_d_3 = AND(stage0_D_3, neg_r)
remainder_0 = SUM(stage0_P_4,masked_d_0,LOW)
rem_co_0 = CARRY(stage0_P_4,masked_d_0,LOW)
remainder_1 = SUM(stage0_P_5,masked_d_1,rem_co_0)
rem_co_1 = CARRY(stage0_P_5,masked_d_1,rem_co_0)
remainder_2 = SUM(stage0_P_6,masked_d_2,rem_co_1)
rem_co_2 = CARRY(stage0_P_6,masked_d_2,rem_co_1)
remainder_3 = SUM(stage0_P_7,masked_d_3,rem_co_2)
rem_co_3 = CARRY(stage0_P_7,masked_d_3,rem_co_2)
```