Strategies for SAT-based Formal Verification

by

Vishnu C. Vimjam

Dissertation submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Dr. Michael S. Hsiao, Chair
Dr. Dong S. Ha
Dr. Sandeep K. Shukla
Dr. R. Michael Buehrer
Dr. Ezra Brown

January 29, 2007

Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Keywords: Learning, Model Checking, Equivalence Checking, ATPG, SAT, BDD

To my loving parents

Basavaiah Naidu and Indira

Strategies for SAT-based Formal Verification

Vishnu C. Vimjam

Abstract

Verification of digital hardware designs is becoming an increasingly complex task as the designs are incorporating more functionality, becoming complex and growing larger in size. Today, verification remains a bottleneck in meeting time-to-market requirements and consumes more than 70% of the overall design-costs. Traditionally, verification has been done using simulation-based approaches, where a set of appropriate test-stimuli is used by the designer. As the designs become more complex, however, simulation-based techniques often fail to capture corner-case errors. Furthermore, unless exhaustively tested, these approaches do not guarantee the correctness of a system with respect to its specifications. As a consequence, formal methods for design verification have been sought after. In formal verification, the conformance of a design to a given set of specifications is proven mathematically, thereby leaving no room for unexplored search spaces. Despite the exponential time/memory complexities often involved within the formal approaches, they have shown promise in capturing subtle bugs, which were missed otherwise.

In this dissertation, we focus on Boolean Satisfiability (SAT) based formal verification, which has gained tremendous importance in the recent past. Importantly, SAT-based approaches often alleviate the memory explosion problem, which had been a bottleneck of the traditional symbolic (Binary Decision Diagram based) approaches. In SAT-based techniques, the set of verification tasks are converted into a set of Boolean formulae, which are checked for satisfiability using a SAT solver. These problems are often NP-complete and are prone to an explosion in the required run-time. To overcome this, we propose novel strategies which utilize both structural and logical information of a sequential circuit. In particular, we devise techniques to extract non-trivial invariants of a design, strengthen properties such that they can be proven faster and interleave bounded reachability analysis with bounded model checking. We provide the necessary algorithms and implementation details in order to automate the proposed techniques. Experiments conducted on a variety of benchmark circuits show that orders of magnitude improvement in overall run-times can be achieved via our techniques compared to the existing state-of-the-art SAT-based approaches.

Acknowledgements

It gives me immense pleasure to thank my advisor Prof. Michael Hsiao for his continued inspiration and guidance throughout my stay at Virginia Tech. Without his profound vision and my endless hours of discussions with him, this dissertation would not have been possible. I am extremely thankful to Prof. Dong Ha, Prof. Sandeep Shukla, Prof. Michael Buehrer and Prof. Ezra Brown for serving on my PhD committee and their valuable suggestions in improving my dissertation.

I am thankful to Sreejit Chakravarty and Srikanth Venkataraman for providing me valuable internship opportunities at Intel Corporation. My discussions with both of them and my supervisors Yi-Shing Chang, Ruifeng Guo and Enamul Amyeen were very helpful and increased my insight to the state-space explosion problem. I would also like to thank Prof. Thomas Martin, Prof. Joseph Tront, Prof. Lynn Abbott and Prof. James Baker for their encouragement and guidance provided during my Masters' days at Virginia Tech.

My sincere thanks to all Proactive Lab colleagues Kameshwar Chandrasekar, Manan Syal, Shuo Sheng, Qingwei Wu, Xiao Liu, Gordon Zhang, Xiaoding Chen, Puneet Gupta, Rajat Arora, Shrirang Yardi, Karthik Channakeshava, Maheshwar Chandrasekar, Ronnie Lajaunie, Bin Li, Danling Chen, Lei Fang, Weixin Wu, Xueqi Chen and Nannan He for all the good times and several valuable discussions during the course of my PhD. I am also thankful to all my room-mates for their support and help during my stay in Blacksburg. Last but not the least, I would like to thank my family, friends and well-wishers for their constant encouragement and support throughout my academic career.

Vishnu C. Vimjam

January 2007

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's computing systems are becoming increasingly complex due to the huge number of features added for realizing high-end applications. These include critical systems such as space applications, medical diagnosis systems, secure banking systems, etc. Digital integrated circuits (IC) form an important component in these systems and range from small end micro-controllers to very complex special purpose processors. As the size and complexity of these IC designs keep on increasing, the ultimate products become increasingly error prone. These errors can be present in the initial design itself or introduced in the middle due to human interference or due to buggy design automation tools. It is quintessential that all the logical or physical errors of an integrated circuit be cleared before the final product is released.

While the verification tasks are applied to both hardware and software of the underlying system, this dissertation will focus mainly on the verification of the hardware. We note that the techniques introduced for bounded model checking, property strengthening, etc., can be incorporated into software verification as well. In the following sections, we will introduce the design flow in VLSI designs and the necessity of design verification. We will talk about some of the issues in verification and point out why verification tasks should be done formally. Finally, we provide the main contributions of this dissertation and provide the overall flow of this document.

1.1 Design Flow

The design process of a system starts with a set of user-defined *specifications*. For example, when designing a traffic-light controller, we require that no two perpendicular sides have the green signal simultaneously. Such specifications are taken into account while designing a system and it is crucial that the system conforms to the *whole* set of specifications. Sometimes, the set of specifications itself might not be complete, which leaves room for unspecified behavior of the system. Such a lack of specification may sometimes be intentional, where the user does not bother about what the system does in those cases.

The set of specifications are first modeled at the behavioral level. Then a structural description at the Register Transfer Level (RTL) is derived. This process is known as *Behavioral* synthesis and is often used for Application Specific Integrated Circuit (ASIC) design. In cases where critical adherence to criteria such as area requirements is needed, designers may directly model structual RTL from the specifications without going through behavioral synthesis. The structual descriptions are then synthesized to gate-level net-lists and then to a lower circuit (transistor/schematic) level. This is known as *Logic* synthesis. The circuit schematic is then converted into a physical layout and a chip is fabricated using these layouts and adding the input-output pads, etc [1].

Besides the transformation of a design from one level to another, it is modified or optimized at each level. Design *modifications* are usually done to add more functionality or to make the design meet the specifications in a stricter sense or to improve the testability of a design *etc*. On the other hand, *optimizations* are done to improve the system's performance, which include reducing the latency/cycle-time, decreasing the area, minimizing the power consumption, etc. In most cases, these optimizations are automated at the RTL and lower levels using computer-aided design tools. Manual optimizations such as those done at the architectural level or placement of functional blocks are also very common [1].

1.2 Design Verification and Complexity Issues

The term *Design Verification* usually refers to various aspects of verification of a design. These include Functional verification, Timing verification, Power verification, etc. In functional verification, the functionality of a design is verified for correctness. In timing verification, the design is checked to see if it satisfies the timing specifications. Note that a design can be functionally correct but might not be producing the outputs at the *specified* time. Similarly, in power verification, a design is checked if it satisfies certain power constraints such as leakage power, maximum instantaneous power, etc.

In this dissertation, we focus on functional verification which is often categorized into two types: (i) Static verification which formally verifies designs with respect to their functional specifications and (ii) Dynamic verification which usually employs simulation based techniques and aims at increasing coverage (such as state coverage, transition coverage etc.) of the test-bench. Dynamic verification is also referred to as Design Validation.

Depending of the size and type of a given functional block, the complexity of a verification task can vary. It also varies according to the kind of specification being verified. For example, it may be non-trivial to estimate the cost-difference of verifying if a violation occurs in *any* clock-cycle compared to verifying if the same violation occurs in a *given* clock-cycle. Verification of a functional block becomes more complex when it is integrated as a part of a bigger system. For example, to check if a divide by zero error occurs in a division unit, we need to check if the output of the unit feeding the divisor can ever be zero. Such tasks are usually referred to as *System-level* verification.

In the worst case, the complexity of a verification task can be *exponential* in time as well as memory which forms the major bottleneck for verification techniques. When the design size is small, verification can be performed fairly easily with the state-of-the-art computing systems. But as a design's complexity or size increases, the time/memory complexity in verification increases exponentially. For example, a design with 30 state-elements can have more than a billion reachable states and checking if a given property holds in the design might require checking all those states. As today's designs have tens to hundreds of thousands of state-elements, methods to reduce the exponential costs of verification is a must.

1.3 Why Formal Verification?

Formal verification deals with establishing a complete proof for verifying a system with respect to its specifications. Conventionally, one can use test-benches (or test-patterns) that are *intuitively* generated by the designer to check if the design has any bugs. However, there are several limitations behind such approaches: (i) Creation and application of test stimuli often requires great expertise both on the part of the design engineer and the verification engineer, (ii) Corner-case errors existing in the implementation might not be covered by the manually-generated test-stimuli, (iii) Automation or manual generation of test stimuli at a full-chip level is very time consuming. As opposed to applying such tests, an exhaustive simulation of all the test vectors can be conducted to cover all the *possible* scenarios. However, this requires simulating a huge number of input patterns which can be impractical even for medium-sized designs. As a result, techniques for formally verifying a system have been sought after in the past. Such techniques are referred to as *Formal Methods*.

Formal methods employ mathematical proof procedures to fully verify a system with respect to its specifications. Some of the underlying engines used in formal methods include techniques that employ Automatic Test Pattern Generation (ATPG), Symbolic engines, and/or Boolean Satisfiability (SAT). Among these, ATPG and SAT based techniques are propositional procedures and try to establish that no counter-example exists for a given verification task. On the other hand, symbolic methods employ graph-based data structures (such as Binary Decision Diagrams (BDDs)) and verification is performed by performing logical operations or quantifications on these structures. Although operations on *ordered* decision diagrams can be performed in linear time with respect to their sizes, the main disadvantage is that the graphs themselves are prone to an explosion in size. In general, ATPG/SAT procedures are prone to an explosion in time, whereas the BDD based approaches are prone to an explosion in memory.

Despite these inherent limitations within formal methods, they have shown much promise in verifying systems in their entirety. For several industrial designs, they were used to find subtle bugs in the past that were not captured via simulation-based approaches. Thus, as a design's size/complexity grows, formal verification becomes *essential* as opposed to *optional*.

1.4 Contributions of this Dissertation

In this dissertation, we focus on SAT-based formal verification approaches with an emphasis on SAT-based Bounded Model Checking [2] and Induction [3] techniques for safety properties. We study various limitations of these approaches in terms of efficiency for safety property checking and propose novel strategies to improve their performance. Below, we present a high-level overview of our contributions whereas the detailed techniques, algorithms and experimental results can be found in Chapters 3 to 7.

There are several limitations in terms of efficiency in the SAT-based Bounded Model Checking and Induction techniques. For checking safety properties, a design is unrolled for several cycles and checked to see if starting from the initial state, the design satisfies the specifications which are written formally as *properties*. For a given property, if nothing can be concluded within an unrolled bound, the bound needs to be increased to check if the property is refuted/proved at a higher bound. As the bound is increased, the Boolean formula generated to be solved by a SAT solver increases linearly. However, even such a linear increase in size leads to an increase the problem complexity exponentially. This directly relates to the efficiency of BMC/Induction approaches. Experiments [4, 5] have shown that these approaches are often successful for properties that can be proven within smaller bounds while they explode in time for properties that require longer bounds to be proven. Existing improvements such as Incremental Learning [6, 7, 8], improved ordering schemes [9, 10] and Simplification techniques [11, 12] etc. improve the efficiency at a given bound but do not have the capability to prove properties at smaller bounds than required otherwise. In this dissertation, we provide automated techniques that aim at proving safety properties at smaller bounds so that the overall runtimes can be significantly smaller. These are described below.

When a circuit is converted to a Boolean formula in Conjunctive Normal Form (CNF), the circuit gates are associated with Boolean variables. As a result, all the logical relationships among the circuit gates also exist in the CNF formula. However, when the Boolean formula is given to a SAT solver, it may not understand these relations, which limits their ability to perform efficient Boolean Constraint Propagation (BCP). In Chapter 3, we present two new techniques to identify two-node sequential logic implications among the gates which can be converted appropriately and added to the Boolean formula [13]. To efficiently

learn these relations, we unroll the circuit a small number of cycles and use our techniques to compute implications that might span more than one cycle. We show that our techniques can learn highly non-trivial implications which when added to the Boolean formula improves the BCP process of the SAT engine and aids in earlier backtracking [13].

Next, we propose novel techniques in Chapter 4 and Chapter 7 to prove *true* properties[1] efficiently. Since Bounded Model Checking aims at *refuting* properties, it suffers while proving true properties, where the design needs to be unrolled very long to complete the proof. This can be overcome by using induction techniques [3] which uses a assume-verify procedure to prove true properties faster. However, the efficiency of induction techniques depends on the respective property at hand. Whereas it proves several properties at smaller bounds, it suffers for those that require larger bounds to be proved. Looking from a different perspective, if we extract illegal state-spaces efficiently and use them as constraints in the induction runs, we can prove harder properties at smaller depths.

In Chapter 4, we propose novel techniques for fast identification of illegal state spaces. We employ ATPG based learning, exploit logic implications and use an enumerative framework to learn more illegal states [14]. Experiments reveal that, when all the learned states are used as constraints at the state-boundaries in induction runs, we are able to prove several true properties that were hard to prove otherwise using induction.

In Chapter 7, we present novel approaches to learn invariants of a design. The main idea here is behind our motivation that hard-to-prove true properties can be proved by knowing information about other true properties [15]. We propose two models from which we extract candidate invariants and prove them using induction. All those candidates that are proved are again added as constraints in induction runs. Experimental results show that we are able to prove further hard-to-prove true properties. This led to orders of magnitude improvements in property checking runtimes [15].

Whereas the techniques mentioned above aid in proving true properties faster, they do not have the capability to refute *false* properties[2] at lower depths. To overcome this limitation, we propose new techniques for strengthening properties and for performing a deeper search for hard properties [16, 17]. In Chapter 5, we

[1] true properties are those that are satisfied by the design

[2] false properties are those that are not satisfied by the design

propose our property strengthening techniques. We employ ATPG-based pre-image computations and co-invariant [3] extractions to efficiently enlarge the property [16]. Resolution and Dynamic variable ordering are further employed to improve the quality of strengthening. Once a property is strengthened, it can be verified using BMC/Induction similar to the original property. Experimental results have shown that the proof-depths can be reduced significantly thereby achieving several orders of magnitude runtime improvements. As opposed to the above techniques, strengthening enables in proving both passing as well as failing properties faster.

In Chapter 6, we present a novel approach that can refute hard *false* properties which require long counter-examples to be violated. The main idea is to follow a sequence of states, in the forward direction from the initial state(s) and in the reverse direction from the target state(s) and employ a bounded model checker to check if at least one backward state can be reached from one forward state [17]. To achieve this, we augment depth-first bounded reachability analysis together with bounded model checking. Such a framework is very powerful because of the exponential number of *new* states that might be covered compared to conventional bounded model checking. While doing so, due to the strengthened target states, true properties can also be proved earlier via induction runs. We further employ controllability-based guiding techniques such that states collected via the depth-first search are tuned toward reaching the targets faster [17]. Comparison with state-of-the-art incremental learning techniques and sequential SAT solvers show that our technique out-performs them in most cases by reaching the target state faster.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows:

- Chapter 2 provides the necessary background to formal verification while discussing in detail about SAT-based techniques, their capabilities and limitations
- Chapter 3 proposes two new techniques for implication learning that aim at improving the efficiency

[3] co-invariants for a property are those that should be invariant if the property is an invariant

of SAT-based Bounded Model Checking

- Chapter 4 presents a novel framework for fast identification of illegal states and their application to enhance SAT-based induction runs
- Chapter 5 presents our property strengthening techniques which enables us to prove properties faster using SAT-based Induction
- Chapter 6 proposes a novel framework that interleaves bounded reachability analysis together with bounded model checking which can reach deeper target states that are hard to reach otherwise
- Chapter 7 presents our static invariant extraction techniques, which enables us to prove harder properties of a design
- Chapter 8 concludes the dissertation with several directions for future work

Chapter 2

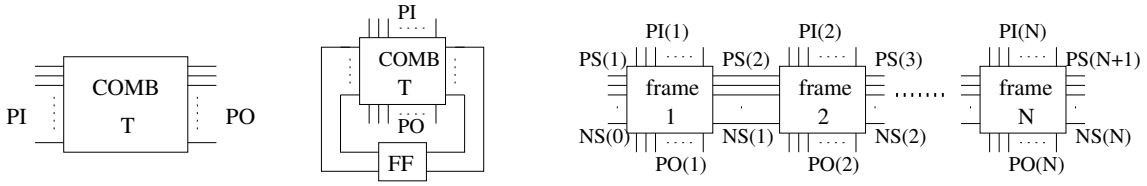
Preliminaries

In this chapter, we provide the necessary background to SAT-based verification techniques. We first introduce the Boolean Satisfiability (SAT) problem, followed by its application to Formal Verification. We then describe all the SAT-based approaches to Equivalence Checking, Unbounded Model Checking, Bounded Model Checking and SAT-based Induction.

2.1 Terms and Notations

Throughout this dissertation, we use the following terms and notations: We represent a combinational circuit as $C =< X, T, Z >$, where X is the set of primary-inputs, T is the combinational logic and Z is the set of primary outputs in the circuit. A sequential circuit is represented as $S =< X, Y, T, Y', Z >$, where X and Z are the primary inputs and primary outputs, respectively, Y, Y' denote the present and next state elements of the circuit and T represents its transition function. A combinational circuit is shown in Figure 2.1(a), a sequential circuit is shown in part (b) of the same figure, and a sequential circuit unrolled for N time-frames (or clock-cycles) using the iterative logic array (ILA) model is shown in Figure 2.1(c). In an ILA model of a sequential circuit, the next-state elements of a given time-frame are connected to the present-state elements of the immediate next time-frame.

Boolean gates are denoted with upper-case alphabets such as $G, G1$, *etc.* We use the set $B = \{0, 1\}$ to represent the Boolean logic values. The lower-case alphabets $v, v1$, *etc.* are used to denote Boolean valued variables and $\overline{v}$, $\overline{v1}$, *etc*, denote their complements. The term *node* is used to represent a logic value assignment to a gate. We represent a node as $G_{v.t}$, where G is a gate, $v \in B$ and $t \in \{1, 2, .., N\}$. In other words, gate G is assigned to value v in time-frame t, where t is limited to the N time-frames. In cases where necessary, we use t for *any* time-frame and not necessarily between 1 and N. For combinational circuits, we represent a node simply as G_v (without the time-frame parameter t). We use the symbol $\wedge$ to refer to the conjunction of the nodes and $\vee$ to refer to their disjunction. We refer to logic simulation as the propagation of logic assignments with the three values $\{0,1,U\}$, where U is the unknown logic value. We use ϕ to represent a property of a design.



(a) Combinational Circuit (b) Sequential Circuit (c) Seq Circuit Unrolled for N time−frames

Figure 2.1. Circuit Notations

Definition 2.1: A gate is said to be *specified* if it is assigned a logic value $v \in B$. If its value is unknown (U), then it is *unspecified*.

Definition 2.2: A specified gate, G, is said to be *unjustified* (by its inputs) if the current assignments of its inputs do not justify the output value of the gate.

For example, let an OR gate G has n fanins $G1$, $G2$, ...Gn. If G is assigned to logic value 1, but none of its inputs are assigned to 1, then G is an unjustified gate. The notion of justification can be easily understood for other gate types.

Definition 2.3: A design is said to be *initializable* if it can be brought to a completely specified state from an all-unspecified initial state, via a *finite* input vector sequence or via an explicit reset signal.

Definition 2.4: A node G_v is said to be *achievable* if starting from an all unknown initial state, there exists

at least one *finite* input vector sequence that can drive a value v at gate G (in some t). Otherwise, G_v is said to be *unachievable*.

Definition 2.5: A property ϕ is said to be *invariant* if that property holds in all the reachable states of the sequential design S. Formally, this is written as $S \models \phi$.

For an initializable design, at least one of G_v or $G_{\overline{v}}$ will be achievable, where $G \in \{$set of gates in the circuit$\}$. If G_v is unachievable, $G_{\overline{v}}$ will be an invariant.

We use the term *logic implication* (or simply implication) to refer to the causal relationship between a set of nodes. An implication $G1_{v1} \wedge G2_{v2} \rightarrow G3_{v3}$ means that whenever the assignment $G1 = v1$ *and* $G2 = v2$ is true, the gate $G3$ cannot be assigned to value $\overline{v3}$.

In the next section, we provide preliminaries to the Boolean Satisfiability problem.

2.2 The Boolean Satisfiability Problem

Boolean Satisfiability (SAT for short) is the first known NP-complete problem. It was proved to belong to the class of NP-complete in 1971 by S. Cook [18].

In the SAT problem, a Boolean formula involving a set of propositional variables is given. This formula can be in Disjunctive Normal Form (DNF) or the Conjunctive Normal Form (CNF). Most of the current day's SAT solvers work on CNFs and hence we will describe it below:

A CNF formula $F =< c, l, v >$ consists of a conjunction of a set of clauses, where

- Each clause c is a disjunction of literals
- Each literal l is a positive or negative occurrence of a variable
- Each variable v is propositional which can take values $\in B$

An example CNF formula consisting of 4 clauses is given below, where $X1, X2, X3, X4, X5$ are Boolean

variables: F = $(X1 \vee X2) \wedge (-X2 \vee X3) \wedge (-X2 \vee X4) \wedge (-X3 \vee -X4 \vee X5)$. The number of literals in each clause is referred to the *size* of the clause. If the size is 1, then it is called a *unit-clause*. A CNF instance is referred to a k-CNF if all the clauses have sizes $\leq k$. Any k'-CNF can be reduced to a k-CNF, where $k' \geq k$, $k \geq 3$, by adding new variables and using them to shorten longer clauses. Whereas SAT problems in DNF form can be solved in polynomial time, k-CNF problems with $k \geq 3$ are NP-complete.

The Boolean Satisfiability problem in the CNF form can be formally stated as follows:

Given a set of clauses C with variables that take value assignments $\in B$, does there exist a satisfying truth assignment to the variables such that all the clauses are satisfied?

Looking from one perspective, this problem is a simplified version of the class of optimization problems (such as ILP). In an optimization problem, we aim at finding a solution such that a given objective is either minimized or maximized. Whereas, in the Boolean Satisfiability problem, we simply search for just *one* arbitrary satisfying solution. Also note that, Boolean Satisfiability is a subset of the general Satisfiability problem in that the variables can only take binary values. There exists several variations of the SAT problem such as MAXSAT, HORNSAT, etc.

One of the initial forms of SAT solving was done using Binary Resolution [19]. This is called the Davis-Putnam procedure. In Binary Resolution, two clauses can be combined and resolved on a common variable that appear in opposing polarities to yield a new clause. This process can be repeated until there are no more such common variables between the two clauses. For example, the above first two clauses $(X1 \vee X2)$ and $(-X2 \vee X3)$ when resolved on $X2$ yields $(X1 \vee X3)$. In the DP-style SAT solving, binary resolution is repeated until we get unit-clauses with each variable having a literal in at least one of these unit-clauses. If these unit-clauses are not consistent (i.e., a variable appears in positive form in one unit-clause and in negative form in another), then that CNF formula is proved to be unsatisfiable. Otherwise, the corresponding truth assignments give a satisfying assignment to the variables. This kind of SAT solving, however, suffers from memory explosion due to the exponential number of new clauses, called *resolvents*, that are derived during the solving process.

This has been improved using a new procedure called DPLL (Davis-Putnam-Love-Longland), where a deci-

sion tree is constructed with the propositional variables. In this style of SAT solving, resolution is performed only implicitly and the truth assignments under any given branch of the decision tree are said to be *implied*. If there is *at least one* path in the decision tree leading to a solution that has consistent variable assignments, then the formula is said to be satisfiable. If no such path exists, then it is unsatisfiable. The pseudo-code for a DPLL style solver is shown in Figure 2.2. Since the decision tree is built over the set of variables with each parent-node having a maximum of 2 children-nodes, this has *exponential* time complexity. However, note that there are no new clauses stored during the process and hence there is no memory-explosion problem.

```
SAT_Solver()
{
    If (atleast one clause violated) return false;
    If (all clauses satisfied) return true and exit;
    (X, v) = MakeNextDecision();   // use a decision heuristic
    Deduce(X=v);       // Boolean Constraint Propagation
    SAT_Solver();  // recurse
    Deduce(X=not(v));   // Boolean Constraint Propagation
    SAT_Solver();  // recurse
    Unspecify(X);
 }
```

Figure 2.2. DPLL Style SAT Algorithm

As seen from the algorithm, the DPLL procedure tries to incrementally build a satisfying truth assignment M for a CNF formula F. M is grown by deducing the truth value of a literal by using already existing assignments in M, or by guessing the value of a variable in F. If a guess for a variable leads to an inconsistency, the procedure backtracks and tries the opposite value. Table 2.1 gives a simple formula and explains step-by-step how the DPLL procedure works.

Most of the modern SAT-solvers are of the DPLL style and are tuned for high-performance. These solvers include, but not limited to, GRASP [20], SATO [21], Chaff [22], BerkMin [23], Siege [24], and MiniSAT [25] etc. The concepts of Conflict-Driven Learning (CDL) and Non-Chronological Backtracking (NCB) have been introduced in the GRASP algorithm [20] which led to a break-through and inspired much of recent developments in SAT solvers. In CDL, upon a conflict, new clauses are extracted by analyzing the nature of each conflict under a given path of the decision-tree. These clauses, when added to the original

Table 2.1. Working of a DPLL procedure

Operation	Assignment	Formula, F	Status
-	-	$(a \vee b)(b \vee \bar{c} \vee d)(\bar{a} \vee \bar{b})(\bar{a} \vee \bar{c} \vee \bar{d})(a)$	start
deduce a	a=1	$(\top)(b \vee \bar{c} \vee d)(\bar{b})(\bar{c} \vee \bar{d})(\top)$	unknown
deduce b	b=0	$(\top)(\bar{c} \vee d)(\top)(\bar{c} \vee \bar{d})(\top)$	unknown
guess c	c=1	$(\top)(d)(\top)(\bar{d})(\top)$	unknown
deduce d	d=1	$(\top)(\top)(\top)(\bot)(\top)$	conflict
undo c	-	$(\top)(\bar{c} \vee d)(\top)(\bar{c} \vee \bar{d})(\top)$	unknown
guess c	c=0	$(\top)(\top)(\top)(\top)(\top)$	satisfied

Assignment set $\{$a=1, b=0, c=0, d=0/1$\}$ is a model for F

clause database, constrain the search space by avoiding entering the same conflict space. Subsequently, it directs the search toward the non-searched regions. Note that due to this addition of clauses, SAT solving via CDL is prone to memory explosion. When compared with the original DPLL algorithm, CDL gives rise to a trade-off between time and memory. However, practical results have shown that the amount of savings in execution time usually outweighs the rather small amounts of additional memory requirements. Furthermore, if the number of conflict-induced clauses is large, ranking of these newly added clauses can be made, and those less-useful clauses may be removed [22, 23].

Other high-performance oriented features of the modern SAT solvers include 2-literal watching [21, 22], efficient decision heuristics [22, 23], efficient clause management [23], better resolution of conflicts [26] and efficient implementation [25], etc. Note that, in spite of all these improvements [27], the complexity of SAT solving is still exponential in the worst case. There has been a tremendous amount of research being conducted these days [28] to identify better SAT solving techniques. Much research has also been dedicated to developing pre-processor engines for SAT such as [29, 30, 31] etc. These engines try to simplify and minimize the CNF instances such that the resulting CNFs can be solved faster using the conventional SAT solvers. Experiments have shown that in many cases significant runtime improvements can be achieved via preprocessing. Whereas most of the practical solvers are general purpose, research is also being done to tune the SAT solvers for specific applications such as verification.

The main frameworks in SAT-based Formal Verification are SAT-based Equivalence Checking and SAT-based Model Checking, which are described in the rest of this chapter.

2.3 SAT-based Equivalence Checking

Equivalence checking forms an important part in the design verification process. Due to erroneous modifications or optimizations to the implementation, bugs can be added into the design. Hence, whenever a design is modified/optimized in any way, it is essential to check it for equivalence with respect to the original design. Currently, equivalence checking tools are employed to verify the equivalence of RTL models to the transformed net-list model (after logic synthesis) or the equivalence of two net-list models implementing the same functionality.

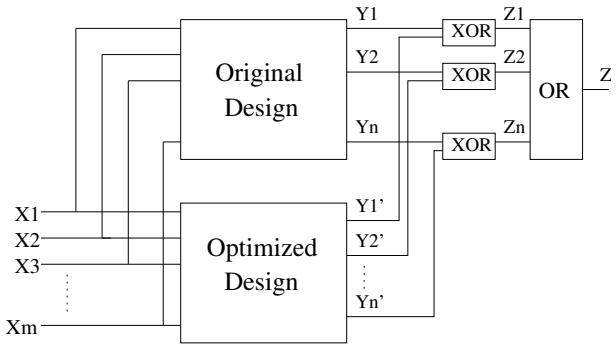


Figure 2.3. CEC Framework

Let C be a multi-output combinational circuit with m primary inputs $X1$, $X2$, ..., Xm and n primary outputs $Y1$, $Y2$, ..., Yn. Let C' be the modified/optimized circuit of C with outputs $Y1'$, $Y2'$, ..., Yn'. To verify that $C \equiv C'$, a miter circuit [32] is constructed as shown in Figure 2.3. In this miter circuit, each corresponding primary output-pair Yi and Yi' from the two circuits are fed to an XOR gate, whose output forms a local miter Zi. All such local miters (Zi) are then fed to an OR gate to obtain the final global miter output Z. The objective of an equivalence checker is to verify that Z is a tautology 0; in other words, to verify $Z = 1$ is unachievable. In SAT-based equivalence checking, the miter circuit is first converted to a CNF formula along with the added constraint clause (Z). If the resulting CNF is satisfiable, then a satisfying solution

exists. Such a solution serves as a counter-example and can be used to debug the design. On the other hand, if the CNF is unsatisfiable, it can be concluded that the optimized circuit is functionally equivalent to the original (golden) one.

As mentioned earlier, a general-purpose SAT solver might not be competitive enough for equivalence checking purposes. Whereas they perform well in easy-to-solve cases, additional learning is necessary to solve the hard instances. Tuning the SAT checks for Equivalence checking has been exploited in several ways. Examples include Robust Boolean reasoning [33], Recursive Learning [34, 35], Exploiting local relations [36], Signal correlation guided solving [37, 38], etc.

While combinational equivalence checking is known to be a coNP-hard problem [39], sequential equivalence checking is considered EXP-space complete [40]. Instead of simply checking the combinational logics for equality, verifying sequential equivalence involves verifying the equality of outputs in *every* reachable state of the product-machine of both the implementation and the golden models.

There exists several notions of equivalence in sequential circuits, which differ in the assumption(s) made with respect to the operation and working environment of the sequential circuit. For circuits with an external reset state, a notion of reset-equivalence is defined, according to which, two sequential circuits are said to be equivalent if their input-output behaviors are identical after reset. Reset equivalence assumes that the external environment is capable of generating the reset signal for both circuits under consideration. However, many real designs do not have an external reset state [41]. For circuits without an external reset, other notions of sequential equivalence have been defined [41, 42, 43, 44]. In [42], the notion of sequential hardware equivalence (SHE) has been proposed, which considers two sequential circuits as equivalent if their input-output behaviors are identical after an initialization sequence has been applied. However, SHE requires the preservation of only one initializing sequence and as a consequence, it can lead to erroneous design replacements if the design environment cannot generate the specific initializing sequence under which the replacement is verified against the original design. In [43], a notion of design replacement in terms of sequential redundancy is proposed. Another classification of sequential equivalence called 3-valued safe replaceability was introduced in [44]. In contrast to the above classifications, the notions of safe replaceability and delay replaceability [41] do not make any assumptions either about the design-environment or

about the steady state behavior of the design (such as initialization). Although safe replaceability provides a more general form of design replacement compared to delay replaceability [41], it is more practical to target verification of delay replaceability of sequential circuits.

Conventional techniques for sequential equivalence checking are based either upon binary decision diagrams (BDD) or upon automatic test pattern generators (ATPG). Due to the recent progress in SAT solvers, sequential equivalence checking using SAT procedures have been explored. Interested readers are referred to [45, 46, 40] for more details on SAT-based sequential equivalence checking.

2.4 SAT-based Model Checking

Model checking deals with the verification of design (or a model) with respect to a given set of specifications (or properties). In model checking [47, 48], the design to be verified is modeled as a Kripke structure (or a finite-state machine) and the properties are written in temporal logic. The reachable states of the design are traversed from an initial state to verify these properties formally.

A property can be of two types [48]: (i) Safety and (ii) Liveness. Safety properties are formulated to verify that nothing bad will happen in the design. These include properties such as *Never x, Always x, After x Never y, After x Always y, After x Always y Unless z, etc*. For example, the condition that no two perpendicular streets should have the green traffic signal simultaneously, is a safety property. Liveness properties are formulated to verify that something good will eventually happen as the system keeps running. These include properties such as *Eventually x, After x Eventually y, After x Eventually y Unless z, etc.* For example, verifying that an acknowledgment eventually occurs after a request is made, is a liveness property.

Early model checking algorithms are *explicit* in the sense that they work directly by traversing the State Transition Graph (STG) of the design. However, as the design size grows, constructing an STG itself will be impractical since the number of states can be exponential. Hence, the practicality of explicit model checking is very limited.

A breakthrough in the form of Symbolic Model Checking (SMC) using BDDs was introduced in [49] to

efficiently automate the model checking process (without needing explicit STGs). Symbolic Model Checking uses efficient Boolean encodings to compactly represent Kripke structures/finite-state machines. By replacing the transition relation and the sets of states with BDD structures, SMC is shown to handle much larger designs. In particular, Boolean function manipulations using ordered BDDs can be performed in linear time [50] with respect to the size of the BDDs. However, an ordered BDD is highly sensitive to its variable order and quickly explodes if a bad ordering is used. Due to this fact, the applicability of symbolic methods remains limited to small/medium-sized designs. Interested readers are referred to [51] for efficient implementation of a BDD package and to [52, 53, 54, 55, 56, 57, 58] for efficient variable ordering schemes for constructing reduced ordered BDDs (ROBDDs).

Due to the advent of high-performance SAT solvers, SAT-based techniques for model checking have recently gained industry-wide acceptance over the symbolic methods. All the tasks that need to be verified, are converted to propositional Boolean formulas which can be checked for satisfiability using a SAT solver. Based on the nature of the model checking approaches, SAT-based Model checking can be further classified into two categories (i) SAT-based Unbounded Model Checking and (ii) SAT-based Bounded Model Checking. We will detail these in the following sections.

2.4.1 SAT-based Unbounded Model Checking

In SAT-based Unbounded Model Checking (UMC), a SAT solver is employed to perform reachability analysis of the design and verify its correctness with respect to a property. Let ϕ be a property to be checked and let S_0 be the set of initial-state(s) of the model M. In property checking, we want to verify if $M \models \phi$. Putting it in a different way, if there exists a path from any initial state to a state that satisfies $\neg\phi$, then we know that the model M does not satisfy the property ϕ.

The two fundamental steps in reachability analysis are Image and Preimage computations.

Definition 2.6: The set of all states that can be reached in one step from a state s forms the Image of s.

Definition 2.7: The set of all states that can reach a state s in one step forms the Preimage of s.

Given a sequential circuit of the form $S =< X, Y, T, Y', Z >$, the characteristic function of the image set for a set of states S can be computed as below:

$Img(s)$ = $\exists_X \exists_Y (S \wedge T)$

Here, the symbol $\exists$ refers to the existential quantification. The right hand side of the above equation existentially quantifies the input variables X and the present state variables Y from $s \wedge T$ and results in a set of states involving the next state variables Y'. This set of states form the image of S. The above equation can be used repeatedly to find the set of states that can be reached in 1, 2, 3, ... clock cycles and so on. Once the set of states reach a fix-point (say, $CompleteImg(s)$), that set represents the complete set of reachable states of the design. This type of analysis is usually called *Forward Reachability Analysis*.

As an example of forward reachability analysis, consider the simple state transition graph shown in Figure 2.4, where $s0$ is the initial state. The states $s1$, $s2$ and $s3$ are reachable from $s0$ in one step and form the image set for $s0$. Similarly, only one state $s4$ is reachable from $s1$ which forms the image of $s1$. Continuing this way in forward direction, we can obtain a fixed-point which consists of all the reachable states that can be reached from the initial state $s0$. It is easy to see from this example, that the design has 9 reachable states while others are illegal (example $s7$ and $s10$).

Similarly, the characteristic function of the preimage set for a state s can be computed using the following equation:

$PreImg(s)$ = $\exists_X \exists_{Y'} (T \wedge S)$

In contrast to the Image operation, note that here we quantify the next state variables Y'. Thus the result is a set of states involving the present state variables Y, which forms the preimage of s. This can be repeated as before to obtain all the set of states that can reach s in 1, 2, 3, ... clock cycles and so on. Once we reach a fix-point (say, $CompletePreImg(s)$), the final result gives the complete set of states that can reach s. This type of analysis is usually called *Backward Reachability Analysis*.

As an example of backward reachability analysis, again consider the STG shown in Figure 2.4. Let us consider a target state $s11$. This state can be reached from $s8$, $s9$ and $s10$ in one step which forms the

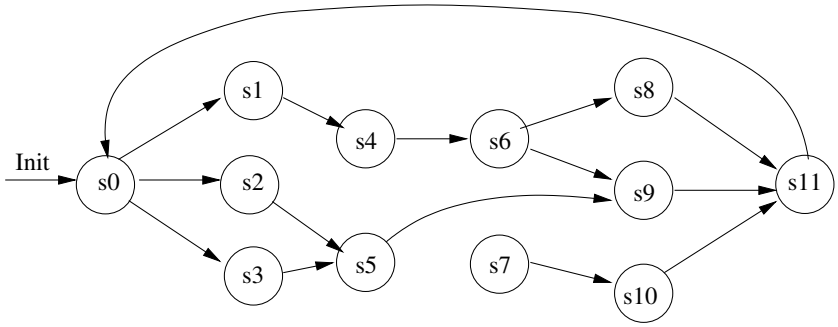


Figure 2.4. Example State Transition Graph

preimage set of $s11$. Similarly, only one state $s6$ can reach $s9$ which forms its preimage set. Continuing this way in the backward direction, we reach a fixed-point which consists of all the set of states that can reach $s11$. If the initial state $s0$ is present in such a set (as does in this example), then we know that there exists a path from $s0$ to the target state $s11$ of the design.

Now consider another target state $s10$. Its preimage set is $\{s7\}$ and the state $s7$ has no predecessors. So we reached a fixed-point with states $\{s10, s7\}$. In this case, we can conclude there does not exist a path from the initial state $s0$ to the state $s10$ of the design thereby learning that $s10$ is illegal (as well as all its predecessors).

From the above examples, we can see that both forward and backward analysis are *complete* procedures themselves and in general there does not exist a clear distinction as to which procedure is more efficient. However, one might judiciously apply either one depending on the underlying engines he is using to perform the analysis or by using his/her intuition about the design. For example, if we know that a design has significantly more reachable states than unreachable states and if a target state is intuitively unreachable, it might be efficient to perform backward reachability, since a fixed-point might be reached faster. Likewise, if we know that the design has significantly more unreachable states than unreachable states and if a target state is intuitively unreachable, it might be better to perform forward reachability.

Similarly, for reachable target states, intuition might be applied. For example, the state $s8$ can be reached via only one path from $s0$. In such cases, performing a backward reachability might be efficient since the

sets of preimage states traversed and stored can be significantly smaller than the image states.

Besides performing only forward or only backward reachability analysis, one can also interleave them [49, 59]. In such cases, certain number of forward and backward steps are computed in each iteration and checked to see if they intersect with at least one state. Note that the efficiency of such an interleaving depends entirely on the target at hand and the nature of the corresponding state transition graph of the design.

In SAT-based UMC, the fundamental image and preimage operations are done using an *all-solution* SAT solver [59,60,61]. Note that a general-purpose SAT solver simply gives us *one* solution (if exists) for a given CNF formula. On the other hand, an *all-solution* SAT solver computes all the satisfying solutions. If we extract the assignments for variables Y' or Y in all these solutions, that would yield the corresponding Image or Preimage. As opposed to the BDD-based computations, the SAT-based procedures do not necessarily rely on canonicity and the set of states reached are simply stored as a set of clauses. To avoid reaching the same solution space again, *blocking clauses* are added which allows for automatic variable quantifications. The clauses obtained at the end of an iteration can be used as the starting states for the next iteration and so on. Experiments [60, 62, 63] conducted have shown that SAT based approaches often complete reachability analysis for large designs which were hard for BDD-based approaches.

To verify if $M \models \phi$, where ϕ is a safety property, we can simply check if the sets $CompleteImg(S_0)$ and $CompletePreImg(\neg\phi)$ has any state in common. If so, then there exists a path from a state in S_0 to a state satisfying $\neg\phi$. This path can be used as a counter-example to debug the design. In practice, the image of S_0 and the preimage of $\neg\phi$ are computed one step at a time and checked for intersection. The number of steps needed for terminating the process depends on the property-at-hand ϕ. If ϕ indeed holds for the design, there will not be a counter-example at any step and the process needs to be continued until a fixed-point is seen in either forward or backward direction.

2.4.2 SAT-based Bounded Model Checking

Bounded Model Checking (BMC) based on SAT procedures [2] is an alternative verification approach as compared to the above Unbounded Model checking. In UMC, the images and preimages are computed

explicitly. Whereas in BMC, we simply search for the existence of a path of *bounded* length between S_0 and those state(s) satisfying $\neg\phi$ (where ϕ is a safety property). To perform this, a Boolean formula is constructed by unrolling the circuit into several time-frames together with a monitor circuit. This constructed formula is satisfiable if and only if the underlying transition system can realize a bounded sequence of state transitions that exposes a path from S_0 to $\neg\phi$.

A typical BMC framework for safety property checking is shown in Figure 2.5 where S_0 represents the initial state(s), T_1, T_2, ..., T_k represent each unrolled transition, and P_1, P_2, ..., P_k represent the property assertions in each unrolled copy. (Here, $P = \neg\phi$ and the monitor circuit is a logical OR function). The total CNF instance can be obtained by concatenating the CNF formulas for S_0, T_0, T_1, ..., T_k, and P_1, P_2, ..., P_k and the monitor circuit. Given such a CNF instance, the SAT solver performs an implicit exploration of the reachable states to verify whether the property assertion is satisfiable within the bound k, from S_0. If the CNF formula for the BMC instance is satisfiable, a counter-example (trace) is generated which exposes an error in the design. If the property is not satisfiable, then the necessity of increasing the bound comes into play.

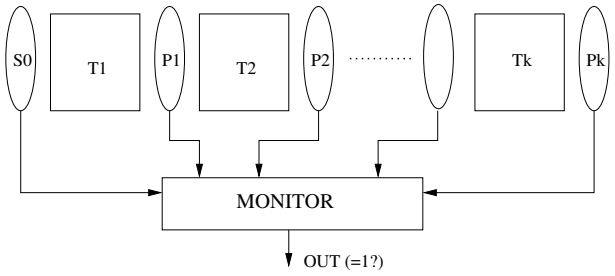


Figure 2.5. BMC Framework for Safety Properties

Note that, similar to the UMC case, the effectiveness of BMC depends on the property-at-hand. If the property can be refuted with a short counter-example (i.e., within a small bound), then BMC is very efficient. On the other hand, if the property is an invariant (i.e., holds forever), we need to increase the bound until the sequential diameter [2] of the circuit is reached. The *diameter* of a design can be defined as the largest simple path in its state transition graph. In other words, it is the maximum distance between any two states s_i and s_j such that s_j is reachable from s_i. Diameter of a design can be exponential in terms of the number

of state-elements and as a consequence, BMC based approaches are limited in practice to only refutation of properties within small bounds. Interested readers can refer to [64] for efficient diameter estimation techniques.

As a simple example, again consider the state-transition graph shown in Figure 2.4. Its diameter is 6 (for example, path $s0$-$s1$-$s4$-$s6$-$s8$-$s11$). Let ϕ be a simple safety property that can only be violated at state $s9$. Initially, we check if the initial state $s0$ can violate ϕ. Since it cannot be violated, we unroll the design for one time-frame and check for violations of ϕ at the next state elements while setting the starting state to $s0$. During decision making, the SAT solver implicitly searches the states $s1$, $s2$ and $s3$ that are the only ones reachable in the first cycle. Since these do not violate ϕ, we further increase the number of unrollments (i.e. bound) to 2 and continue the process. Finally, when we reach a bound of 4, we can reach the state $s9$ and the SAT solver produces a counter-example (either $s0$-$s2$-$s5$-$s9$, or $s0$-$s3$-$s5$-$s9$). On the other hand, if we were given a property ϕ that can be violated only at states $s7$ or $s10$, we would not have found a counter-example even as the bound is increased until the diameter of 6. In such cases, we can conclude that ϕ is an invariant.

Whereas BMC can be used straightforward to check safety properties, liveness property checking is slightly more complex. For example, to check a liveness property *Eventually x*, we need to verify that all paths from the starting state lead to *x*. In other words, if there does not exist a infinite path that goes through only the bad states (i.e. those violating *x*), then the liveness property holds. Such a check can be formulated using BMC [2]. Initially, we set bound to 1 and check if a state satisfying $\neg x$ can be reached in one cycle. If so, then not all paths satisfy *x* in one cycle and we need to increase the bound to 2. While doing so, we assert that $\neg x$ holds in cycle 1 and check if a state satisfying $\neg x$ can be reached in cycle 2. If so, we can repeat the process by increasing the bound. On the other hand, at any given bound, if the SAT check is unsatisfiable, then that means there does not exist an infinite path that goes only through the bad states and hence we can conclude the liveness property holds. For more details on SAT-based Liveness checking, readers are referred to [2, 65].

Since BMC aims at solving Boolean formulae using propositional SAT procedures, it enjoys several advantages and at the same time has several limitations. The main advantages of BMC can be formulated as follows: (i) Linear conversion time to CNF formulae especially for hardware designs. This is because once

a translation is obtained for a single unrollment of the circuit, it can simply be replicated without additional analysis. (ii) SAT-solving techniques have taken a plunge and several high-performance SAT solvers such as zChaff [22], BerkMin [23], Siege [24], MiniSAT [25] *etc* are available (iii) Incremental learning possible and helpful in many cases. This is because of the replicated nature of the clauses in BMC instances. When a BMC instance is constructed for an unrolled sequential circuit, most of the clauses in the combinational logic are replicated while retaining the already existing combinational logic clauses. Thus, learning conflict clauses while solving smaller instances, and carrying them over while solving larger instances can be greatly beneficial in solving the larger instances efficiently. (iv) No serious memory explosion problem especially at smaller depths.

BMC despite being practical for providing counter-examples for false properties, suffers from two main limitations: (i) The technique is theoretically incomplete unless the bound is increased until the diameter of the system. In other words, BMC suffers heavily while solving true properties. As mentioned above, finding diameter is also another reachability problem. (ii) Explosion in time at higher depths. This is because, as the bound is increased, the number of variables and clauses increase linearly. However, even such a linear increase leads to exponential increase in solving complexity. Thus BMC is efficient for solving properties that can be refuted via smaller counter-examples.

To overcome the above limitations, several approaches have been proposed in the past: These include Incremental learning [6,7,8], Improved ordering strategies [6,8,66,9], BDD-based learning mechanisms [67,68], and CNF simplification techniques [11, 12]. Furthermore, when a BMC instance is proved unsatisfiable using a SAT solver, we can learn the UNSAT core[1] from the SAT engine. This UNSAT core information is exploited in various ways [69, 70, 71] to efficiently abstract the design according to the property-at-hand.

2.4.3 SAT-based Induction

In contrast to the above UMC and BMC approaches, Induction [72, 3, 73] is a special technique that aims at proving invariant properties without performing state-space traversals. For example, in [72], equivalence of

[1] An UNSAT core is a subset of the clauses of the CNF which itself is unsatisfiable

certain internal signals is assumed in initial clock cycles and it is checked to see if they remain equivalent forever in all clock cycles. The main advantage here is that, if those equivalences are indeed true, the two circuits might be proved very quickly for equivalence. Below, we provide details on SAT-based induction whereas the same theory applies to Induction, in general.

Let S_0 be the initial state(s) of a finite-transition system with T as its transition relation. Let ϕ be a property to be verified. When the system is unrolled to an ILA of length k, let T_i and ϕ_i respectively denote the transition relation and the property assertion at step i respectively. Then, the Boolean formula for the complete unrolled instance with the property can be constructed by

$$\phi^*(1, 2, .., k) \wedge T^*(1, 2, .., k), \text{where}$$

$\phi^*(1, 2, .., k) = \phi_1 \wedge \phi_2 \wedge .. \wedge \phi_k$ and $T^*(1, 2, .., k) = T_1 \wedge T_2 \wedge .. \wedge T_k$. In SAT-based induction [3], the base case $S_0 \rightarrow \phi$ and the induction step $\phi_1 \wedge T_1 \rightarrow \phi_2$ are translated to Boolean formulas $\neg(\neg S_0 \vee \phi)$ and $\phi_1 \wedge T_1 \wedge \bar{\phi}_2$, respectively. These two CNF formulas are checked for satisfiability using a SAT solver. If the base case formula is satisfiable, then one can readily conclude ϕ is not an invariant. Otherwise, the induction step is performed. If it is unsatisfiable, then ϕ can be concluded as an invariant. Otherwise, nothing can be concluded about ϕ. For most properties, however, this *simple induction* scheme is insufficient, and a stronger induction scheme called *induction with depth* can be used. In this new scheme, the base case is modified to $S_0 \wedge T^*(1, 2, .., k) \rightarrow \phi^*(1, 2, .., k)$, and the induction step is modified to $\phi^*(1, 2, .., k) \wedge T^*(1, 2, .., k+1) \rightarrow \phi_{k+1}$, both of which are again converted into respective CNF formulas. If nothing can be concluded about ϕ at a depth k, it is increased to a higher depth k' and the SAT checks are repeated.

To make the procedure complete, the search needs to be limited to simple paths. This is conducted by adding *unique state constraints* [3], which restrict that no state can be repeated in the satisfying solution. For example, consider a simple STG with two *unreachable* states $s1$ and $s2$ and two transitions $s1 \rightarrow s1$ and $s1 \rightarrow s2$. Now, consider a property ϕ that can only be violated at $s2$ i.e. ϕ is invariant. If we conduct induction step runs for ϕ, then we obtain a counter-example at any depth n, such that the first $n - 1$ states are $s1$ and the last state is $s2$. However, if we use unique state constraints, then $s1$ cannot repeat in the counter-example and we would be proving the property at depth 2 itself.

For finite-transition systems, when the unique-state constraints are added, there always exists a finite k such

that the formula for the induction step is unsatisfiable [3] for invariant properties.

The main motivation behind induction is to prove invariants quickly at smaller depths. All the invariants that are proved via induction are called *inductive invariants*. In general, UMC or BMC based approaches are good for quick refutations of a property, where as induction based approaches are better for proving a property to be an invariant.

2.5 Summary

In this chapter, we provided a formal background to the Boolean Satisfiability problem followed by the SAT-based Formal Verification techniques. In particular, we outlined the SAT-based Equivalence Checking and SAT-based Model Checking frameworks while describing the state-of-the-art enhancements that have been done to make these approaches more efficient.

Chapter 3

Increasing Deductibility in SAT Instances

In this chapter, we will first explain the drawbacks of SAT-based Bounded Model Checking framework, followed by the related work done in the past. We then describe our new technique to alleviate this problem along-with experimental results and some directions for future work.

3.1 Motivation

First and foremost, remember that if all the BMC instances up to a bound k are not satisfiable, then nothing can be deduced about the property. If the property-at-hand is an invariant, we might need to go until the *diameter* of the system to prove it. This diameter can be exponential in terms of the number of state-elements and hence we need to reduce the execution time at each bound as much as possible.

Secondly, note that every time we increase the bound, new variables and clauses are added to the CNF formula. Though this increase is linear in number with respect to the bound, the corresponding increase in solving complexity is exponential in time. For example when going from a bound of k to k', the complexity increases from $2^{n.k}$ to $2^{n.k'}$, where n is the number of gates in the sequential circuit.

For the above two reasons, BMC becomes inefficient as the bound is increased. In practice, BMC is used to

identify as many counter-examples (bugs) as possible within a given small bound. If no counter-examples are found for a property within that bound, UMC techniques might be employed to proceed further.

Thirdly, when a circuit is converted into a CNF formula, all the circuit structural information is lost. In other words, given a CNF formula, a SAT solver cannot identify which variables are primary inputs, which ones are flip-flops etc. Sometimes, such information might be significantly helpful in order to make better decisions or for exploiting Boolean reasoning etc.

3.2 Related Work

Recently, several techniques have been developed that try to learn information from the circuit. Whereas some of these are general, some are specifically aimed at improving BMC. In [68], approximate reachability analysis using BDDs is performed and the BDDs obtained at each state-boundary are converted into clauses to constrain the SAT engine. Since BDDs are employed, care must be taken to avoid excessive pre-processing times as well the number of clauses added. Another BDD-based analysis is proposed in [67] to extract constraint clauses statically or dynamically. In order to restrict both memory and computational costs, the learning is limited only to local regions.

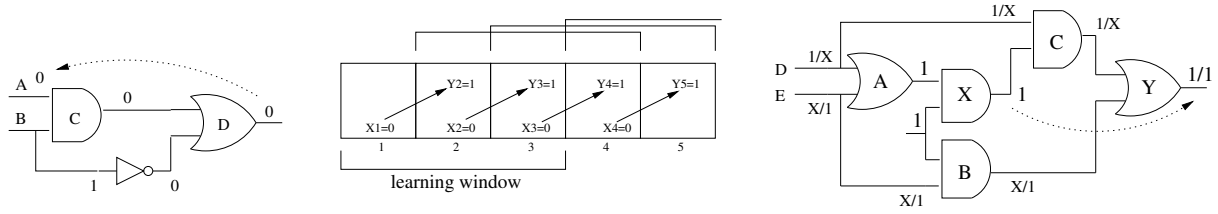
Generalized rules for deducing relations among variables have been proposed in [36] on the basis of unit-propagation. However, the analysis is also limited locally to reduce cost. In Simp2C [74], indirect implications learned via logic simulation are converted to constraint clauses to speed up the SAT solving in BMC. In the example shown in Figure 3.1(a), logic simulation of D_0 results in A_0, and hence the clause $(D \vee \neg A)$ is learned, which helps the SAT engine to learn contra-positively that BCP(A_1) results in D_1. Besides such learning, Simp2C proposes the replication of the learned clauses throughout the entire BMC bound. A simple replication is shown in Figure 3.1(b), where an implication $X1_0 \rightarrow Y2_1$ learned in a small window is replicated throughout [1].

A similar form of learning was employed in the Hypre [29] preprocessor. For example, if clauses $(H \vee \neg A)$, $(H \vee \neg B)$, $(H \vee \neg C)$ and $(A \vee B \vee C \vee D)$ exist in the original CNF, a new clause $(H \vee D)$ is learned via

[1] Here, $X1$ refers to signal X in frame 1 and $Y2$ refers to signal Y in frame 2 and so on

hyper-resolution. Besides this, Hypre performs equality reduction. If two clauses ($A \vee \neg B$) and ($\neg A \vee B$) exist in the CNF, then variables A and B are equivalent and one of the variables can be eliminated by replacing with the other. Similar reductions can be done for complement variables $[(A \vee B) \wedge (\neg A \vee \neg B)]$ and constant variables $[(A)$ or $(\neg A)]$.

Since only logic simulation (or BCP) was employed, the learning of both Simp2C and Hypre is limited. Non-increasing variable elimination has been proposed in NiVER [30], in which variables are eliminated via resolution if doing so do not increase the size of the overall CNF formula. Although Hypre and NiVER are proposed for general-purpose SAT solving, they are equally applicable for SAT-based BMC.



(a) Indirect **(b) Replication in several windows** **(c) EBL**

Figure 3.1. Example Implication Learning

Next, we describe the Extended Backward learning (EBL) procedure developed in [75], which was originally used for the identification of untestable faults. This uses the input justification scenarios of an unjustified gate to learn new implications that are common in all the justification scenarios. We explain the procedure for an AND gate which can be extended to all other gate types: Let G be a AND gate and let $Y_v \rightarrow G_0$ hold. When $Y = v$, G becomes unjustified and let $X1$, $X2$, ..., Xn be its n unspecified inputs. Now, $G = 0$ can be justified by a logic 0 assignment on any of these n inputs. In such a case, the intersection of gate assignments obtained by logic simulating the node combinations ($Y_v \wedge X1_0$), ($Y_v \wedge X2_0$), ..., ($Y_v \wedge Xn_0$) are new implications of the node Y_v. Note that the assignment set $\{(Y_v \wedge X1_0) \cap (Y_v \wedge X2_0) \cap ... \cap (Y_v \wedge Xn_0)\}$ automatically contains the assignments that can be obtained by logic simulating Y_v alone. Hence the implications obtained through EBL are strictly a super-set of those obtained via simple logic simulation (Simp2C or Hypre). We will illustrate EBL via an example circuit shown in Figure 3.1(c). Initially, we have $X_1 \rightarrow A_1$ which makes gate A unjustified. Performing EBL using input justification scenarios for $A = 1$ helps us to learn $X_1 \rightarrow Y_1$. Via the contra-positive law, the implication $Y_0 \rightarrow X_0$ is

also learned. Our experience indicates that these contra-positive implications often aid in significant increase in the total number of non-trivial implications learned from the circuit. In the example of Figure 3.1(a), A_0 is the only justification scenario for C_0 (given D_0) and hence we can automatically conclude $D_0 \rightarrow A_0$ (*i.e.* no explicit intersection is necessary).

Though the learning obtained through EBL procedure is efficient, it might not be adequate for BMC purposes. We conjecture that the more non-trivial [2] a learned relation is, the more it will help a SAT engine in reducing the solving times during BMC. To improve the learning, we propose two new learning strategies that demonstrate superior performance, which can be applied along with [68] or [67] for additional performance benefits.

3.3 Increasing the Deductibility for SAT-based BMC

In this section, we describe new learning techniques based on static implication learning and binary resolution for enhancing SAT-based BMC. First, consider the following definition:

Definition 3.1: *Deductibility* of a CNF formula is defined as $D = \sum_{X} |BCP(X_0)| + \sum_{X} |BCP(X_1)|$, where $X \in \{\text{variable set of the CNF}\}$ and $|BCP(X_v)|$ denotes the number of other truth assignments (i.e. implications) due to assignment $X = v$.

The main motivation behind our work is to add non-trivial clauses to a CNF instance which increases the deductibility in the final CNF formula. Clearly, the newly added clauses will be helpful only if the SAT engine cannot deduce them through BCP from the original CNF. In this regard, we explore the underlying circuit structure to deduce non-trivial relations and add them to the original CNF instance. Our techniques are specially applicable for BMC or sequential ATPG problems, where the large CNF formulas can overwhelm the SAT solvers, making *learning* critically beneficial. Since the implications learned through EBL are more non-trivial than those learned from either Simp2C or Hypre, we have developed our two strategies by extending the concept of EBL. As such, our learning is a superset of the learning provided by EBL.

[2] Non-triviality of a relation is related to how many decisions a SAT solver takes to learn it

3.3.1 Learning via Justification Enumeration

Let S represent the set of implications in the intersection of two opposing nodes related to a gate X, *i.e.*, X_0 and X_1. Consider S to be non-empty, and let G_v be a node in S. Since G_v is implied by both X_0 and X_1, according to the dilemma rule [76], G is a constant gate achievable to only one value v. However, if the circuit has no gates with constant values, then no such sets would exist. Our aim is to find *conditional* constants, whereby under the presence of some other node Y_v, the set S is no longer empty for a target gate X. First, we consider the following lemma which is a conditional variant of the above dilemma.

Lemma 3.1: *If the intersection of implications of two mutually exclusive nodes X_0 and X_1 is non-empty under a condition Y_v, then the nodes in the intersection are implications of Y_v. In other words, if ($X_0 \wedge Y_v$) $\rightarrow Z_{v1}$ and ($X_1 \wedge Y_v$) $\rightarrow Z_{v1}$, then $Y_v \rightarrow Z_{v1}$.*

Proof: The proof can be obtained simply by applying binary resolution. Resolving on gate X eliminates the nodes X_0 and X_1 from the two conflicting sets ($X_0 \wedge Y_v \wedge Z_{\overline{v1}}$) and ($X_1 \wedge Y_v \wedge Z_{\overline{v1}}$), thus resulting in the new conflicting set ($Y_v \wedge Z_{\overline{v1}}$) which means $Y_v \rightarrow Z_{v1}$. $\diamond$

We call Y_v as the *base* node, gate X as the *resolver* and X_0, X_1 as the *resolving* nodes. Given Lemma 1, one can aim at finding base nodes such that the intersecting set S of the resolving nodes is non-empty. However, in order to avoid selecting too many base nodes for a given resolver gate, we formulate this problem in the reverse fashion to find good resolver gates for a given base node. Recall that during the application of EBL for a node Y_v, we find unjustified gates and use their unspecified inputs for justifications. We find that such unspecified inputs for an unjustified gate are good *candidate resolver* gates for the base node Y_v. This heuristic selection helps in (1) capturing efficient re-convergences of the resolving nodes through the base node, (2) increasing the specified region around the unjustified gate, and (3) keeping the simulation overhead as small as possible since half of the work is already being done during EBL computation. Since the nodes X_0 and X_1 create similar local simulation events, we bit-pack [3] them during simulation to further reduce the overhead. Though simpler in theory, we have observed that this formulation can extract several powerful

[3] In bit-packing, one bit of a word is allocated to each case split such that all case splits can be simulated simultaneously

relations that cannot be captured by a SAT engine (through BCP) or by Simp2C or the EBL procedure.

We demonstrate this type of learning with the help of an example shown in Figure 3.2. Let the implications $E_1 \rightarrow C_0$, $E_1 \rightarrow H_1$, $D_0 \rightarrow F_0$, $G_1 \rightarrow I_1$, $K_0 \rightarrow J_1$ be those relations already learned from the circuit (shown by dotted lines). Let X_0 be the base node. Its implications are simply A_0, B_0 and K_0 from the circuit shown in the figure. None of these implications can further imply any other gate. Consider gate E which is one of the unspecified inputs of A. According to Lemma 1, we can compute the set $\{(Y_v \wedge E_0) \cap (Y_v \wedge E_1)\}$ as the new implications of Y_v. Besides Lemma 1, we can see that the node combination $(Y_v \wedge E_1)$ leads to a conflict around gate B during logic simulation. Thus the only possible assignment for E under condition X_0 is a logic 0 and we conclude $X_0 \rightarrow E_0$ and hence $E_1 \rightarrow X_1$ via the contra-positive law. We note that these relations cannot be derived with EBL even if it is iterated until its convergence point (*i.e.,* when no more implications can be deduced from the circuit). Such internal relations can be very useful, and adding them to the implication graph enables us to capture even more non-trivial implications during the learning process.

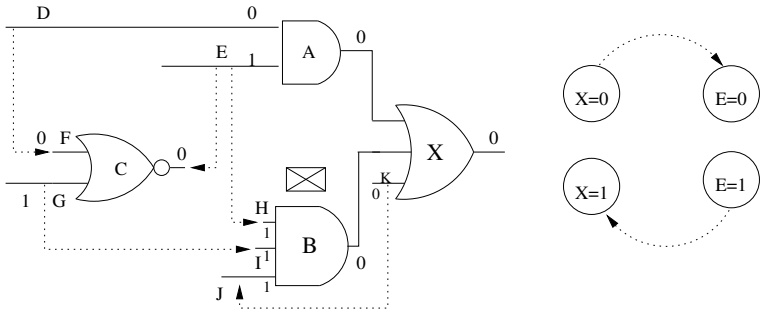


Figure 3.2. Example Learning via Justification Enumeration

In Lemma 1, we have described the case where only a single gate is used for resolution under a base node. We now extend Lemma 1 to consider multiple gates under a base node. For instance, in the example just discussed for Figure 3.2, when gate A becomes unjustified for node X_0, multi-gate resolution can also be performed using the three new justification scenarios $(D_0 \wedge E_0)$, $(D_0 \wedge E_1)$ and $(D_1 \wedge E_0)$. Note that these three scenarios automatically cover the direct binary resolutions on the unspecified inputs of A, and they represent the complete justification scenarios for $A = 0$. Even though such an enumeration would intuitively be more powerful than *EBL + Lemma 1*, it would be expensive for performing several logic simulations.

For an unjustified gate with n unspecified inputs, $2^n - 1$ simulations (corresponding to each of the $2^n - 1$ justification scenarios) would be necessary to check if the intersection has any nodes in common. However, for the case when $n = 2$, only three simulations are needed. Hence, we resort to such an exhaustive enumeration only when $n = 2$ and employ Lemma 1 along with EBL for cases where $n > 2$ to keep the pre-processing time low. As before, all justification scenarios can be bit-packed and simulated together to enable faster simulation. We incorporated this kind of learning as the first strategy (*JEnum*) of our tool. Note that these implications cannot be captured by the propositions in [77] because of two main reasons: (1) the result is limited to only identifying equivalent variables where as ours can identify one-way implications as well and (2) no concrete procedure is used for selecting variables for *double unit-propagation look ahead*, where as ours uses the justification scenarios for unjustified gates which increases the specified region within the circuit. Similarly the work in [78] is limited to only equivalence reasoning. In the next subsection, we describe an improved learning strategy that has more extraction capability than *JEnum*.

3.3.2 Learning via Justification Frontier

Consider Figure 3.3(a). Node A_0 is a direct backward implication of the node X_0. Since neither gate B nor gate C is specified to logic 0, gate A is an unjustified gate. Learning based on EBL would use the justification scenarios $B = 0$ and $C = 0$. However, gate B is not a fanout stem and hence no new implications would be learned for X_0, using EBL. We extend the justification scenarios for gate A to its Immediate Justification Frontier (which is comprised of the complete set of nodes farthest away from gate A that would individually justify assignment $A = 0$).

The Immediate Justification Frontier (IJF) of a gate's assignment can be obtained by taking the immediate input justification scenarios for that assignment and recursively extending them backward until all the gates achieve their non-controlling values or a primary input is reached. For example, using Figure 3.3(a) again, extending the immediate justification scenarios for $A = 0$ would give the node-set $\{C_0, D_1, F_1, G_1\}$ as the IJF. Extending in such a way has two main advantages: First, any fanout stems along the backward path (for example, C, D and E in Figure 3.3(a)) would be covered automatically. Secondly, the number of implications of the nodes in the IJF would be greater than the number of implications of nodes in the

immediate input justification scenarios. Hence, this would lead to capturing more complex relations that might not be obtained even when *JEnum* learning is enumerated for $n > 2$. According to this new procedure, the new gate assignments in the set $\{(X_0 \wedge C_0) \cap (X_0 \wedge D_1) \cap (X_0 \wedge F_1) \cap (X_0 \wedge G_1)\}$ are also implications of X_0 in Figure 3.3(a).

Again, we consider the application of Lemma 1 here, which can be directly applied to the nodes in the IJF of the unjustified gate. Similar to *JEnum*, any node in the IJF and its opposite node can be bit-packed and simulated together to reduce the simulation time. However if an unjustified gate has m nodes in its IJF, then we would need $2 \times m$ bits for simulation using Lemma 1. In general, m can be much larger than the number of inputs (say, n) of the unjustified gate. In such cases, the pre-processing time can be very high. To avoid this and to obtain even more non-trivial learning, we propose a new technique by modifying the above procedure, which is explained below.

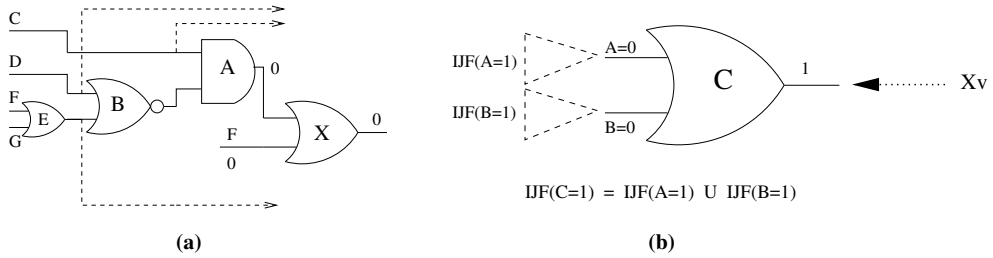


Figure 3.3. Example Learning via Justification Frontier

Consider Figure 3.3(b). Let the set IJF(A=1) contain i nodes $A1_{v1}$, $A2_{v2}$, ..., Ai_{vi} and the set IJF(B=1) contain j nodes $B1_{v1}$, $B2_{v2}$, ..., Bj_{vj} respectively. Note that the set IJF(C=1) is the union of the sets IJF(A=1) and IJF(B=1). Let $X_v \rightarrow C_1$ which makes gate C unjustified. According to Lemma 1, using input A, we would compute $\{(X_v \wedge A_0) \cap (X_v \wedge A_1)\}$. Replacing A_1 with its IJF, we would like to compute the set $\{(X_v \wedge A_0) \cap (X_v \wedge A1_{v1}) \cap (X_v \wedge A2_{v2}) \cap ... \cap (X_v \wedge Ai_{vi})\}$. Note that this set represents the complete possible assignment cases for A, and hence assignments in this set are implications of X_v. Similarly, using all possible assignment cases for B, we can compute the set $\{(X_v \wedge B_0) \cap (X_v \wedge B1_{v1}) \cap (X_v \wedge B2_{v2}) \cap ... \cap (X_v \wedge Bj_{vj})\}$ which can be added as implications of node X_v. Apart from these, the assignment set $\{(X_v \wedge A1_{v1}) \cap (X_v \wedge A2_{v2}) \cap ... \cap (X_v \wedge Ai_{vi}) \cap (X_v \wedge B1_{v1}) \cap (X_v \wedge B2_{v2}) \cap ... \cap (X_v \wedge Bj_{vj})\}$ (using the IJF for C_1) are also implications of node X_v. We incorporated this kind of learning as our second

strategy (*JFron*) in our tool. Note that this new procedure requires only $m + n$ bits for simulation (m for IJF(C=1), n for (A_0, B_0)) to process one unjustified gate as opposed to $2 \times m$ needed earlier. At the same time, since nodes A_0 and B_0 have more implications than each of $\{A1_{\overline{v1}}, A2_{\overline{v2}}, ...Ai_{\overline{vi}}\}$ and each of $\{B1_{\overline{v1}}, B2_{\overline{v2}}, ...Bj_{\overline{vj}}\}$ respectively, this will provide superior learning as compared to the direct application of Lemma 1 on IJF(C=1). However, the enumeration step done in *JEnum* may sometimes lead to new implications that cannot be obtained through *JFron*. Whether *JEnum* extracts more non-trivial implications or *JFron* does, depends on the underlying circuit structure.

3.4 Efficiency Issues

We have integrated the above two learning strategies into our tool Static Extraction Techniques for SAT-based BMC (SET-SAT), which computes the sequential implication graph of a circuit and then generates the final CNF instance for BMC. Converting a learned implication into a clause is straight-forward. For example, the four implications $X_0 \rightarrow Y_0$, $X_0 \rightarrow Y_1$, $X_1 \rightarrow Y_0$ and $X_1 \rightarrow Y_1$ would be converted to the clauses ($X\ \neg Y$), ($X\ Y$), ($\neg X\ \neg Y$) and ($\neg X\ Y$) respectively. Note that adding a clause for an implication automatically takes care of the contrapositive implication as well.

There are three important implementation issues involved in SET-SAT. The first is concerned with the ordering of nodes for which the static logic implications are learned. Initially, we have only direct implication edges. More edges are learned as the learning process progresses. Experience from other researchers [75] as well as our own experiments support that learning in a levelized fashion (*i.e.*, breadth-first manner) leads to best case of learning. Given a gate-level net list, we first levelize it and compute the implications for the nodes in a levelized manner starting from the inputs level to the outputs level. Secondly, since we add the learned implications as extra clauses, the BMC formula sizes can grow very large. Similar to [79], we employ a sequential implication graph for efficient implication storing (refer to [79] for its construction details) and then convert the implications to clauses. However, the final implication graph obtained during the learning can have many redundant edges. For example, edge $X_v \rightarrow Z_v$ will be redundant if the edges $X_v \rightarrow Y_v$ and $Y_v \rightarrow Z_v$ already exist in the graph. Hence, we first process the implication graph to remove all such redundant edges and then create the final CNF formula. Note that *deductibility* remains the same

before and after the edge reduction process, but the new number of clauses added to the BMC formula can reduce significantly. Another implementation issue concerns with the number of iterations (sweeps across the circuit). In general, we have observed that only a small fraction of implications are newly learned after the first iteration. Hence we limit our tool to only one iteration so that the pre-processing time is effective for overall BMC run times.

The overall flow for SET-SAT is as follows: First the implication graph is computed for a circuit using a user-defined window size and then redundant edges are removed to make the graph efficient. The CNF formula for the given initial state(s), property, and the expansion bound is then generated to be solved by a SAT solver.

3.5 Experimental Evaluation

Experiments for circuits from the ISCAS89 and ITC99 benchmark suites were conducted using the zChaff SAT solver (version 2004.11.15) on a Pentium-4 3.2GHz machine having 1GB RAM and running RedHat Linux operating system. We have taken few safety properties for each circuit and selected the top two properties which are hard for zChaff with the original CNF. We ran all of the available techniques and ours on these hard safety properties. Our main aim is to show the relative robustness of our learning as compared to the existing techniques.

Table 3.1 shows the deductibility information in the pre-processing phase. For each circuit, we show the number of gates in column *Gates* and the learning window size used by *win*. We used a window size of 5 for all circuits except for the large circuit s35932 for which 3 was used. We report the *deductibility* numbers (for the middle time-frame variables among *win*) obtained via the original CNF and our two strategies under the columns *Orig*, *JEnum* and *JFron* respectively. Apart from the circuit b10, learning via *JFron* always resulted in higher deductibility as compared to that of *JEnum*. Next, we show our average improvement in deductibility over the original CNF in the column *X*. As seen, deductibility can be increased up to 4.9 times using our learning. Note that this improvement is from the non-trivial implications learned and can lead to greater speedups in BMC instances, as seen next.

Table 3.1. Improvement in Deductibility

Circuit	Gates	win	Orig	JEnum	JFron	X
s400	194	5	13.5K	39.5K	39.9K	2.9
s420.1	253	5	32.2K	112K	113K	3.5
s444	211	5	15.2K	46.6K	48.5K	3.1
s510	243	5	28K	103K	109K	3.8
s526	223	5	18.8K	61.6K	63.6K	3.3
s820	331	5	86.3K	215K	227K	2.6
s832	329	5	86K	217K	226K	2.6
s967	462	5	96.2K	315K	315K	3.3
s9234.1	5883	5	1.43M	3.52M	3.59M	2.5
s13207.1	8803	5	7.32M	13.7M	13.9M	1.9
s15850.1	10533	5	4.2M	8.4M	8.6M	2.0
s35932	18148	3	8.95M	9.94M	9.94M	1.1
b10	206	5	15.2K	67.6K	66.8K	4.4
b11	770	5	55.1K	271K	272K	4.9
b12	1076	5	139K	490K	501K	3.6
b13	362	5	17.5K	59.8K	60K	3.4

win: learning window size X: Our Average / Orig

Table 3.2. BMC results for safety properties for ISCAS and ITC Benchmarks

Circuit	k	Prop	Orig	NiVER [30]		Simp2C [74]		EBL [75]		SET-SAT				X_{avg}
										JEnum		JFron		
		(SAT)	T_t	T_p	T_t	T_p	T_t	T_p	T_t	T_p	T_t	T_p	T_t	
s400	200	1(S)	19	0.02	6.6	0.01	17.3	0.1	2.9	0.3	4.0	0.4	**1.2**	7.2
		2(U)	15	0.02	0.9	0.01	10.2	0.1	1.4	0.3	**0.7**	0.4	0.8	20
s420.1	200	1(U)	393	0.02	40.8	0.03	55.3	0.2	29.2	0.5	20.5	0.8	**9.8**	26
		2(U)	144	0.02	10.9	0.03	103	0.2	32.2	0.5	33.5	0.8	**8.6**	7.8
s444	200	1(S)	74	0.03	10.5	0.03	20	0.1	14.5	0.3	5.6	0.5	**5.1**	13.8
		2(U)	28	0.03	7.5	0.03	31	0.1	1.1	0.3	**0.7**	0.5	2.1	20
s510	200	1(S)	223	0.02	256	0.02	380	0.2	5.0	1.6	**1.8**	3	3.5	84
		2(S)	1284	0.02	156	0.02	1136	0.2	409	1.6	11.6	3	**3.5**	170
s526	200	1(S)	1033	0.03	678	0.02	1063	0.5	47.5	3.2	**18.2**	6.7	23.7	48.3
		2(U)	833	0.03	342	0.02	541	0.5	16.5	3.2	**4.4**	6.7	8.7	127
s820	200	1(U)	511	0.04	134	0.03	489	0.5	294	3.1	27.4	6.7	**8.6**	28
		2(U)	317	0.04	59	0.03	266	0.5	193	3.1	14.1	6.7	**9.8**	26
s832	200	1(U)	610	0.04	118	0.03	542	0.5	339	3.3	56.3	6.6	**19.1**	16
		2(S)	272	0.04	44.7	0.03	190	0.5	30.5	3.3	19.8	6.6	**10.4**	18
s967	200	1(U)	236	0.05	84.3	0.05	235	1.1	22.7	3.4	**7.6**	8	9.9	27
		2(U)	148	0.05	46.6	0.05	146	1.1	11.6	3.4	**6.91**	8	9.1	18.5
s9234.1	60	1(S)	5893	0.5	3388	0.53	4589	18	1296	32	**36.2**	46	148	64
		2(S)	4712	0.5	1566	0.53	3821	18	49	32	**33**	46	46.5	118
s13207.1	100	1(U)	852	0.5	287	1.9	154	37	**55**	132	141	112	117	6.6
		2(U)	701	0.5	285	1.9	157	37	**68**	132	142	112	116	5.4
s15850.1	80	1(S)	2155	0.6	274	1.2	949	36	281	73	217	108	**174**	11
		2(S)	2559	0.6	317	1.2	1085	36	601	73	564	108	**252**	6
s35932	80	1(S)	-TO-	1.0	**1539**	8.4	6301	103	2510	145	2544	149	2546	>4
		2(S)	4796	1.0	**848**	8.4	4160	103	1026	145	1068	149	1072	4.4
b10	150	1(U)	172	0.03	52	0.01	123	0.2	59.2	0.9	30.9	1.1	**14.4**	7.5
		2(U)	208	0.03	74	0.01	123	0.2	202	0.9	23.9	1.1	**18.1**	9.8
b11	150	2(S)	502	0.1	237	0.03	75	1.7	8.3	3.4	**5.7**	6	6.4	83
		2(S)	593	0.1	185	0.03	318	1.7	27.7	3.4	**4.5**	6	7.2	111
b12	100	1(U)	1566	0.2	696	0.05	1392	2.6	486	18	461	24	**324**	4
		2(U)	1262	0.2	631	0.05	1158	2.6	449	18	**229**	24	394	4
b13	150	1(U)	35	0.1	14.8	0.01	28.2	0.2	18.6	0.4	**2.5**	0.6	3.3	12
		2(U)	40	0.1	10.1	0.01	28	0.2	13.1	0.4	2.4	0.6	**2.0**	18

T_p: Preprocessing time

T_t: Total execution time (includes T_p)

X_{avg}: Our average speed-up (of *JEnum* and *JFron*) over *Orig*.

TO: Instance timed out after 10000 seconds.

Table 3.2 reports the actual BMC experimental results. An all-zero initial state is used for all circuits. For each circuit, *k* gives the BMC bound used, which was selected based on the circuit size and its complexity. The column *Prop(SAT)* shows the property number followed by its satisfiability (*S* for SAT and *U* for UNSAT). The columns *Orig*, *NiVER*, *Simp2C*, *EBL*, *JEnum* and *JFron* report the times taken by zChaff with the original CNF, [30], [74], [75] and our two strategies, respectively. The sub-columns T_p and T_t report the pre-processing and total times (T_p + solving time) respectively in seconds. For a fair comparison with NiVER (since it works on an entire unrolled instance where as all others work on a learning window), we used T_p of NiVER as (original $T_p \times win/k$). The column X_{avg} shows our average speed-up (of *JEnum* and *JFron*) obtained over *Orig*. As seen from the table, our learning strategies can significantly decrease the overall BMC run times. For most circuits, a consistent speed-up of more than an order of magnitude was achieved over the original BMC run. Since our learning enables the deduction of more implied literals at each decision, it increases the efficiency of the SAT-solver for both satisfiable as well as unsatisfiable instances. Next, we will compare our tool with NiVER [30]. Although NiVER is very fast in reducing the CNF formula, the overall improvement NiVER achieves over the original CNF is limited for most circuits. Except for circuit s35932, our tool consistently outperformed NiVER for all other circuits. Similar to NiVER, SIMP2C's preprocessing cost is also low. However, the main drawback is that most of the added clauses help the SAT engine in only one direction whereas the other direction incurs additional BCP costs. Though our tool adds more clauses compared to Simp2C, the clauses are highly non-trivial and the resulting deductive power is superior. For many circuits, our tool outperformed Simp2C by an order of magnitude or more.

Finally, consider our performance with respect to EBL. It can be seen from Table 3.2 that EBL performs better for some instances, although more implications can be learned using *JEnum* and *JFron*. One possible reason for this is that the newly added clauses increase the correlation among the CNF variables and the number of clauses each variable appears in. In some cases, the benefit gained due to the added learning might be lost due to a (possibly) inferior ordering followed by the SAT engine. Note that even in such cases, our tool performs much better than *Orig*, thus revealing the robustness of the proposed learning strategies. For circuit s13207.1, although the total times for EBL are better, the actual solving times are smaller for *JEnum* and *JFron*. For most other circuits shown in the table, adding the learned non-trivial relations to the

original CNF instance has shown consistent improvements over EBL.

Let us now look into the differences between *JEnum* and *JFron*. As seen in the theory, the learning obtained by *JFron* is in general superior than *JEnum* (though strictly not a superset). However, *JFron* requires more computational cost in the pre-processing phase. For most of the cases where *JEnum* outperformed *JFron*, the actual solving times differed only marginally. With an intuition in the design and the property being checked, a verifier can choose *JFron* for hard properties and *JEnum* for reduced learning and pre-processing times. For large industrial circuits, the CNF instances generated for BMC can be huge and hence both of our learning strategies might be very effective in reducing the total solving times.

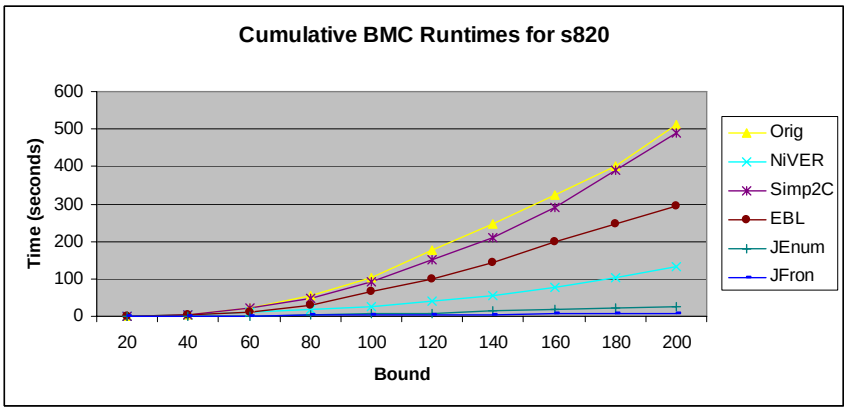
In Figure 3.4, we show example cumulative runtimes for property 2 for circuit s820 and for property 1 for circuit s9234.1. The main intention here is to demonstrate the superiority of our learning techniques compared with others. As we can see, the cumulative runtimes for the original BMC run, run with NiVER, Simp2C and EBL increase drastically as the bound is increased, whereas via our techniques we were able to contain the runtimes even at larger depths. This shows the efficiency of our techniques in enabling deeper BMC runs, under given resource limits.

3.6 Summary

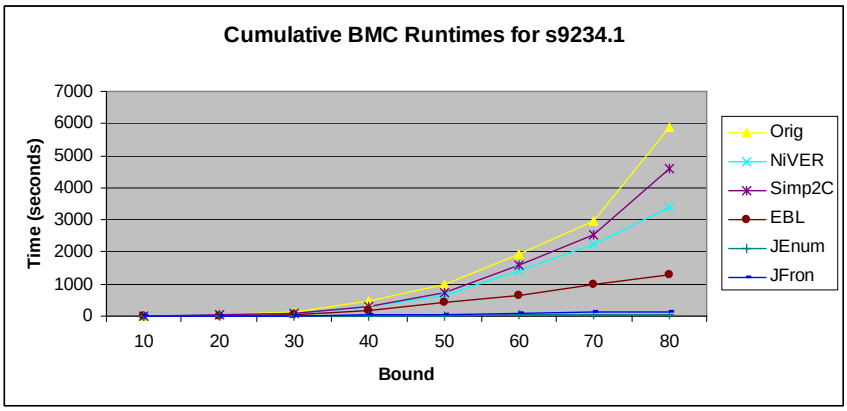
In this chapter, we have presented new and efficient learning techniques based on powerful combination of binary resolution and static implication learning to extract non-trivial relations from a circuit. These sequential logic implications act as constraint clauses and turn away the SAT engine from reaching unwanted regions. Experimental results show that our learning increases the deductibility and hence provides consistent performance improvement over a wide variety of circuits and properties. Our approach can be used as a pre-processing step for the SAT-based BMC problem and can complement dynamic/property-specific techniques.

There are several directions possible for future work. First, note that in some cases, the solving times actually increase due to the added clauses. This might be because, due to the added clauses, the number of occurrences of each variable is increased, which leads to a possibly different decision order chosen by the

SAT solver. It will be definitely interesting to study the effect of added clauses upon the variable order and find a better ordering strategy. Secondly, all the clauses being added might not be useful in the same way. For example, some clauses might be trivial and can be easily learned by the SAT solver, given the other clauses. Careful elimination of such clauses might reduce the burden on the SAT solver and avoids overheads in BCP. Finally, new ways to learn non-trivial implications would definitely be a promising future-work.



(a) Cumulative BMC runtimes for s820



(b) Cumulative BMC runtimes for s9234.1

Figure 3.4. Example Cumulative BMC Runtimes

Chapter 4

Illegal State Identification

In this chapter, we will first explain the limitations within the SAT-based induction framework, followed by outlining the work done in the past. We will then describe a novel framework developed to quickly extract the illegal (i.e. unreachable) states of a sequential circuit, which are used to improve the SAT-based induction runs. We also provide experimental results based on our framework and identify potential future research areas.

4.1 Motivation

There is a main drawback for SAT-based induction, which is the lack of knowledge about the illegal state-space of the design. This limitation manifests itself in two ways. First, since the SAT-solver relies on a branch-and-bound procedure, the SAT engine may make poor decisions and later learn that the decisions lead to a functionally impossible (i.e. unreachable) space. This problem is critical in an induction step run since the initial state space is not constrained to any reachable state(s). Hence, the knowledge of illegal states before-hand can constrain the SAT-search. Second, if the formula for the induction step at a depth k is satisfiable, the satisfying solution forms a simple path s_1, s_2, ..., s_{k+1} such that the first k states satisfy ϕ but the last state s_{k+1} does not. In such a scenario, two cases are possible: (i) the first state s_1 is a reachable state, and hence ϕ is definitely not an invariant. (ii) the first state s_1 is an illegal state in which case the

counter-example obtained is spurious and nothing can be concluded about ϕ. However, it is very difficult to identify the reachability of s_1.

Due to the reasons discussed above, SAT-based induction runs can be improved by learning as many illegal states as possible. Since a thorough reachability analysis (even over-approximate) can be prohibitively expensive, we try to gather under-approximate illegal state information of a design, via an alternative, low-cost analysis.

4.2 Related Work

Whenever a counter-example (as explained above) is obtained in the induction step run, the authors in [80] assume s_1 to be an illegal state and obtain a strengthened property $\phi \wedge \neg s_1$. The main intuition behind it is that the new property is stronger than the original one and might be proved at an earlier depth via induction. However, if the induction run has several such counter-examples (which can be exponential in the number of state elements), the new property obtained might not be strong enough.

Another technique for improving induction runs has been explored in the form of Incremental learning [6, 7, 8]. Due to the incremental nature of the induction runs, the idea here is to identify relevant conflict clauses that can be carried over to the successive iterations. As a consequence, the effectiveness of these techniques depends largely on the learning ability of the underlying SAT solver. Another limitation is that all the conflict clauses that are *local* to the property constraints cannot be carried to higher bounds.

4.3 Fast Illegal State Extraction

In this section, we describe our techniques for learning illegal states. We use the term *time-frame* to represent one transition relation of a sequential circuit. Let B=$\{0, 1\}$ represent the set of Boolean logic values. The notation $X_{v.\tau}$ is used to represent a signal X set to value $v \in B$ in a given time-frame τ. If the parameter τ is not relevant, we simply use X_v. Also, we assume all the designs are initializable (either by an initialization sequence or via an explicit reset signal). This ensures that the system has only one terminally strongly

connected component (TSCC) [1].

We refer to logic simulation as 3-valued logic propagation done using the three values $\{0, 1, U\}$, where U is the unknown logic value. A signal X is said to be *specified* if it attains a known logic value $v \in B$; otherwise it is *unspecified* (or unknown). A composite signal assignment, $F = \{X1_{v1} \wedge X2_{v2} \wedge ... \wedge Xn_{vn}\}$, where $v1, v2, ..., vn \in B$, is said to be *achievable* if starting from an all unknown initial state, there exists at least one *finite* input vector sequence that can set $F = 1$ in a finite time-frame τ (*i.e.* to specify $X1 = v1$, $X2 = v2$, ..., $Xn = vn$ in τ). Otherwise, F is said to be *unachievable*. For an initializable design, at least one of F or $\neg F$ will be achievable. If F is unachievable, $\neg F$ will be an invariant.

4.3.1 Learning Using An ATPG

In this subsection, we will explain our ATPG based illegal state identification technique. The following definition forms the basis of our learning.

Definition 4.1: (**n-Cycle-Unreachable**) A state s is said to be n-cycle-unreachable if the n^{th}-level preimage of s is empty.

In other words, if there does not exist any state that can reach s in n cycles, then s is unreachable. Note that this definition is a simpler form of the complete induction seen in chapter 2. If a state is n-cycle-unreachable, it is also (*n+1*)-cycle-unreachable, and so on. We employ a combinational ATPG-based method (using the PODEM[2] algorithm [81]) to quickly identify a subset of such states. The main advantage is that the learned information can be used repeatedly to constrain all the state-spaces without any additional analysis. Another added benefit is that, states that can be reachable only from the learned unreachable states can also be avoided by the SAT-solver during its search.

We unroll a circuit for n time-frames (similar to an Iterative Logic Array expansion) as shown in Figure

[1] A TSCC comprises a set of states of a design such that any state in the TSCC is reachable from all other states in the TSCC and once the design enters a state in the TSCC, it remains in the TSCC until powered off

[2] PODEM backtraces from a given objective (say, X_v) and makes decisions on the primary inputs to justify it (*i.e.* to set $X = v$)

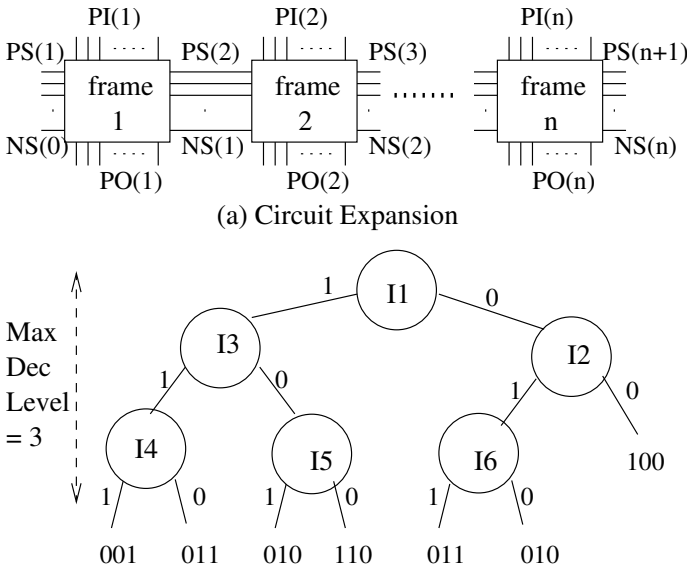


Figure 4.1. Learning via ATPG

5.1(a), where the unrolled frames are numbered 1, 2, ..., n. The lines PI(i), PO(i), PS(i), NS(i) represent the primary inputs, primary outputs, present-state elements, next-state elements for frame i respectively. All the present-state elements for frames 2 to n are simply treated as buffers. We can embed a state s at PS($n+1$) and try to justify it while treating the state elements at PS(1) as pseudo-primary inputs (PPIs). In such a scenario, if no input assignment can justify the state s, it is n-cycle-unreachable. This is because, allowing the flip-flops at PS(1) to be fully controllable assumes all the states are possible at that boundary and hence guarantees $\neg s$ is an invariant. However, pre-image computations usually have exponential complexity and it will be infeasible to compute them even for a small subset of states. In order to identify unreachable states in an efficient manner, we propose a *state-independent* procedure as explained below.

Without loss of generality, consider the flip-flops at PS($n+1$) be arranged in an order $\{X1,\ X2,\ X3,$ .., $Xn_{ff}\}$, where n_{ff} is the number of flip-flops. Our main goal is to perform an ATPG search without targeting any particular state. We start with the first flip-flop $X1$ set to an objective logic value (say, 1) and start the search process (*i.e.* justifications via PODEM). If $X1_{1.n+1}$ is justified during the decision process, we proceed to justify $X2$ to a value (say, 0) and continue. In this way, the entire decision tree is constructed

like a free-BDD[3] as shown in Figure 5.1(b), where the nodes represent the decision variables ($I1$, $I2$, .., $I6$ can be any primary input or a PPI) and the leaves indicates the logic values attained by the flip-flops at PS($n+1$).

Consider a group of 3 flip-flops ($X1$, $X2$, $X3$) at PS($n+1$) were *all* specified at *all* leaves of the decision structure. Figure 5.1(b) shows some example values. It can be seen that the partial-states 000, 101 and 111 were not achieved at any of the leaves. Since the decision tree is complete for *specifying* all these 3 flip-flops, these three states can be concluded as illegal. In other words, $\{(\overline{X1} \wedge \overline{X2} \wedge X3) \vee (\overline{X1} \wedge X2 \wedge \overline{X3}) \vee (\overline{X1} \wedge X2 \wedge X3) \vee (X1 \wedge \overline{X2} \wedge \overline{X3}) \vee (X1 \wedge X2 \wedge \overline{X3})\}$ is an invariant. One way of strengthening this invariant is by increasing n, due to which more illegal states can be ruled out. Another way is to eliminate the already known illegal paths (and the corresponding leaves) of the decision structure. For example, if $I1$ and $I2$ of Figure 5.1(b) are PPIs and if we know that $\{I1_0 \wedge I2_0\}$ is an illegal partial-state, then $\{X1_1 \wedge X2_0 \wedge X3_0\}$ is also an illegal partial-state. This is because the partial-state 100 can be obtained via only illegal paths of the free-BDD (in this case, only one such path). In our implementation, we store the illegal states learned in the iterations $n = 1, 2, .., i-1$ and use them to eliminate the illegal paths of the free-BDDs in the iteration $n = i$. All the illegal partial-states accumulated till the end of the last iteration N (user-defined) are converted into constraint clauses.

Algorithm 1 shows our learning algorithm for a given n. Due to space limit, we do not show the above invariant strengthening technique. To keep the computational cost low, we limit the maximum decision level (MDL) to a user-defined value (currently set to 20). After each ATPG run, we obtain a free-BDD from which unreachable partial-states are extracted. We use a technique similar to that of [82] to form the free-BDD. We refer the reader to [82] for details. All the flip-flops that are specified at *all* the leaves of the free-BDD are removed from the order and are not considered in the remaining ATPG runs. Within a given MDL limit, if no such flip-flop exists, we forcefully remove the first flip-flop in the order. This is done to ensure that the algorithm terminates within a maximum of n_{ff} calls to the ModifiedPODEM() procedure.

We use the SCOAP measures [83] to identify if a flip-flop is most controllable to logic 0 or logic 1. Heuristically, we always try to justify a flip-flop to its most controllable value. This is done to specify a value at that

[3] a BDD where all the paths need not have the same variable order

Algorithm 1 n-Cycle Unreachable Learning Algorithm

```
1: Given n, MDL; Unroll circuit n times
2: Order flip-flops at PS(n + 1) and unmark them
3: Compute SCOAP values (C0, C1) for each flip-flop
4: while (not done) do
5:     currDL=0; Initialize all signals to value U
6:     ModifiedPODEM(); // build a free-BDD
7:     LearnIllegalStatesFromFreeBDD();
8:     Mark flip-flops for which free-BDD is complete
9:     If all flip-flops are marked, then done
10: end while
11: ModifiedPODEM()
12:     if (currDL=MDL) then return;
13:     currDL = currDL+1;
14:     X=Next unspecified flip-flop in the order
15:     if (C0(X) ¿ C1(X)), then obj=0; else obj=1
16:     (pi,val)=Backtrace(X,obj);
17:     LogicSim(pi=val);
18:     ModifiedPODEM(); // recurse
19:     LogicSim(pi=not(val));
20:     ModifiedPODEM(); // recurse
21:     LogicSim(pi=U);
22:     currDL = currDL-1;
23: return
```

flip-flop with as few decisions as possible (recall the MDL limit). The MLP procedure [84] is used to obtain the initial ordering of the flip-flops. This procedure computes the input supports for the flip-flops and clusters the ones with closer supports. For our purpose, note that an initial ordering is important, because, once a flip-flop is specified, the next flip-flop that we choose should be co-justifiable easily. The overall complexity of the above algorithm is $O(n_{ff}.2^{MDL})$, which can be be adjusted according to the user-specified value for MDL.

Dynamic Regrouping. Since MLP uses only structural analysis to order flip-flops, we have observed that the ordering may not always help us in justifying a suitable group of flip-flops together. For example, if flip-flops $X1$ and $X2$ are adjacent in the order produced by MLP, it is possible that within MDL, there exists several paths of the free-BDD, at whose leaves $X1$ is specified but not $X2$. This happens when the flip-flop $X2$ does not group well with $X1$ and more decisions are needed along those paths to specify $X2$. In such cases, we will not learn anything illegal involving $X1$ and $X2$. We use a procedure to dynamically regroup the flip-flops as our analysis progresses. Essentially, if we were not able to specify $X2$ at the leaf of a path p during the decision process, we check if any other flip-flops in the lower order have been specified at *all* the leaves of the previously constructed $p-1$ paths and as well at the current path p. If so, all such flip-flops are moved to the position next to $X1$, and $X2$ is moved down the order. If no such flip-flop exists, however, we continue as shown in our regular algorithm (see Algorithm 1). The main intuition behind this dynamic approach is to group flip-flops which are closely related in the Boolean space rather than via circuit structure. The final ordering obtained at the end of unroll depth $n-1$ is used as the initial ordering for the next iteration n.

At each leaf, L, of a free-BDD built in Algorithm 1, we obtain a set of assignments to the flip-flops at PS($n+1$). Let this be partial-state $F1$. Let the flip-flop assignments along the path of the free-BDD for leaf L form a partial state $F2$. We take the intersection of the assignments in $F1$ and $F2$ (say, $F3$) and store them as a list, $LIST$. All such stored partial-states are used in our unified framework as explained in section 4.3.4.

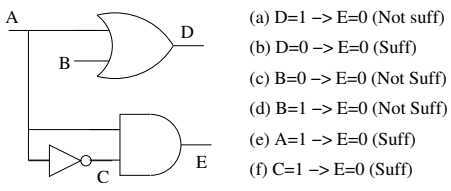


Figure 4.2. Sufficient Implication Examples

4.3.2 Unachievable Partial-State Learning

In this subsection, we explain our necessary assignment looping theory to learn additional unachievable partial-states. First, consider the concept of a general logic implication, which describes the logical dependencies among the signals in a circuit. For example, the implication $X_{0.\tau} \rightarrow Y_{1.\tau+1}$ means that whenever signal X is set to logic 0 in frame τ, the only possible value at signal Y in frame $\tau + 1$ is a logic 1. We define the concept of a *sufficient* implication as follows.

Definition 4.2: (**Sufficiency**). Starting at an all unknown initial state, let I be an input vector sequence of length τ. The implication $X_{v.\tau} \rightarrow Y_{w.\tau-k}$ ($v, w \in B$, $k \geq 0$) is termed *sufficient* if any input vector sequence I that implies $X_{v.\tau}$, also implies $Y_{w.\tau-k}$. Formally, $\forall I : I \rightarrow X_{v.\tau} \Rightarrow I \rightarrow Y_{w.\tau-k}$.

Not all the logical implications inside a circuit are sufficient. For example, consider the simple circuit shown in Figure 4.2 where E_0 is a constant assignment. The implication $D_1 \rightarrow E_0$ holds true but is not sufficient. This is because D_1 can be achieved by an input vector $\{A = U, B = 1\}$ which doesn't cause E_0 via logic simulation. The figure shows a few other sufficient and non-sufficient example implications.

Next, we use the above sufficiency condition and define the following lemma for obtaining unachievable assignments to flip-flops.

Next, we use the above sufficiency condition and propose the following theorem for obtaining unachievable partial-states.

Theorem 4.1: *A partial-state* $F = \{X1_{v1} \wedge X2_{v2} \wedge .. \wedge Xn_{vn}\}$, *(*$v1$, $v2$, .., $vn \in B$*), consisting of* n *flip-flop*

assignments is unachievable, if for any τ, the following sufficient implication exists: ($X1_{v1.\tau} \wedge X2_{v2.\tau} \wedge ... \wedge Xn_{vn.\tau}$) $\rightarrow$ ($X1_{v1.\tau-k} \wedge X2_{v2.\tau-k} \wedge ... \wedge Xn_{vn.\tau-k}$), $k > 0$.

Proof: We prove this by contradiction. Starting from an all unknown initial state at frame 1, let a finite input vector sequence I of length l achieves $F = 1$ in frame l. According to our sufficient implication above, I should also set $F = 1$ in frame $l - k$, which in turn requires $F = 1$ in frames $l - 2k$, $l - 3k$, ..., $l - nk$ and so on, where $n \rightarrow \infty$. For some value of n, $nk \geq l$ which means that F has to be true in a frame ≤ 1. This, however, requires I to have a length $> l$.

[(length(I) = l) $\rightarrow$ (length(I)$> l$) $\Rightarrow$ (length(I) = ∞)].

In other words, no finite input vector sequence can achieve F. $\diamond$

For example, let for any τ, $(X_{0.\tau} \wedge Y_{1.\tau}) \rightarrow (X_{0.\tau-1} \wedge Y_{1.\tau-1})$ be a sufficient implication. Then, by the contrapositive law, $(X_{1.\tau-1} \vee Y_{0.\tau-1}) \rightarrow (X_{1.\tau} \vee Y_{0.\tau})$. In other words, flip-flops X and Y can be initialized and remain only in the following states: $\{00, 10, 11\}$. Note that Theorem 1 holds true only if the implication is a *sufficient* one. Otherwise, we might not conclude unachievability. For example, if the combinational logic shown in Figure 4.2 is a part of a sequential circuit, then $E_{0.\tau} \rightarrow E_{0.\tau-k}$ holds true for any τ, any $k > 0$. However, it *cannot* be a *sufficient* implication according to definition 2. Simply put, an invariant cannot sufficiently imply itself backward. Otherwise, it will not be an invariant.

In a naive manner, we can inject a partial-state onto the circuit, imply its necessary logic values and check if it satisfies the criteria given in Theorem 1. However, checking this for all possible partial-state assignments is impractical even for medium-sized designs. While Theorem 1 is proposed with respect to partial states, it is equally applicable for any composite signal assignment in a circuit. Hence, we apply it efficiently during the static implication computation process and in our unified framework, as explained in the following sub-sections.

4.3.3 Exploiting Logic Implications

Since the circuit information is encoded into a Boolean formula for performing the induction runs, it becomes beneficiary to extract non-trivial relations existing among the circuit signals (*i.e.,* those that cannot be deduced by the SAT solver directly through Boolean Constraint Propagation(BCP)) and inject them as learned clauses into the formula. This has been put to use in [74], where the implications learned via direct logic simulation are converted to clauses for enhancing the SAT runs. In our framework, we make use of the Extended Backward learning (EBL) procedure described in [75]. There are two main advantages behind this: First, the learning procedure used in [74] might miss many complex relations existing inside a circuit. Although the time consumed by the EBL procedure can be higher than that of [74], it has the potential to identify new non-trivial relations. Second, the unachievability theory described in the above sub-section can be efficiently employed during the EBL computation process. Note that the use of EBL is not a main contribution of this paper, but the means of applying our technique via EBL is.

Algorithm 2 Algorithm for Unachievability Learning

1: Given N(odd), Unroll circuit N times; $n = (N + 1)/2$
2: Order signals in frame n from PIs to POs
3: for each signal X in order, for each v in $\{0,1\}$ do
4: LogicSim($X = v$)
5: UG = Set of unjustified gates due to $X = v$
6: for each gate G in UG with controlling value cv do
7: for each unspecified fanin F_i of G do
8: S_i = Set of assignments due to LogicSim($X = v$, $F_i = cv$)
9: if ($X = v, F_i = cv$) hold Theorem 1, add ($X = v$)→($F_i = \overline{cv}$)
10: end for
11: Add ($X = v$)→$\bigcap S_i$ // EBL step
12: end for
13: if ($X = v$) holds Theorem 1, store $X = \bar{v}$ as invariant
14: end for

Algorithm 2 shows our unachievability learning algorithm. Given N, the circuit is unrolled for N time-frames. Each signal assignment X_v ($v \in B$) in the middle frame n is logic simulated and the set of

unjustified gates[4] due to X_v are identified. If G is an unjustified gate with controlling value cv, each of its unspecified fanin Fi is set to value cv and logic simulated together with X_v. The EBL procedure (step 11) computes the intersection of signal assignments in all such logic simulations and stores them as implications of X_v. As soon as the logic simulation of $X_v \wedge Fi_{cv}$ is done, we check if it holds the criteria of Theorem 1. If so, the implication $X_v \rightarrow Fi_{\overline{cv}}$ is learned. At the same time, the contrapositive relation $Fi_{cv} \rightarrow X_{\overline{v}}$ is also learned. Note that all such implications learned via Theorem 1 are highly non-trivial since a SAT engine might not be able to deduce them during its decision making process. In our implementation, we store all the implications learned and use them to identify more signal assignments during logic simulations (at steps 4 and 8). The advantage of doing so is to learn more non-trivial implications when the algorithm proceeds for the remaining signals.

Finally, after all the implications are computed for a given signal assignment X_v, we check if X_v satisfies the criteria of Theorem 1. If so, $X_{\overline{v}}$ is stored as an invariant. All such invariants and learned implications are converted to clauses. For example, for a learned implication $X_0 \rightarrow Y_1$, the clause $(X \vee Y)$ is created. The original EBL algorithm has quadratic complexity in terms of the number of signals of the circuit. Checking our unachievability criteria only requires a small fraction of the run-time needed by EBL.

Figure 4.3 shows 2 time-frames of an example sub-circuit. The copy of a gate X in frame n is named $X^{'}$ in frame $n-1$. The implications $X1_{0.n} \rightarrow {H^{'}}_{1.n-1}$ and $F_{0.n} \rightarrow I_{1.n}$ already exist in the circuit (shown by dotted lines). For simplicity, we do not show all the gates in both the frames. Consider the assignment $E_{1.n}$ due to which the gate C (with logic value 0) is unjustified. The two possible justifications for $C_{0.n}$ are $J_{0.n}$ and $F_{0.n}$ (since both fanins J and F are unspecified currently). When $F_{0.n}$ is logic simulated along with $E_{1.n}$, the figure shows how the assignments ${F^{'}}_{0.n-1}$ and ${E^{'}}_{1.n-1}$ are also implied. Since we are implying the necessary values in the circuit, these are sufficiently implied and hence we can conclude $\{E_1 \wedge F_0\}$ as unachievable according to Theorem 1. Encoding this as a clause $(\neg E \vee F)$ into the induction formula helps in rejecting states that can cause these unachievable signal combinations. In the example shown, $\{X1_0 \wedge X2_1 \wedge X3_1\}$ is one such illegal state.

[4] a gate is *unjustified* if it has a specified value but the current logic values of its fanins do not justify it

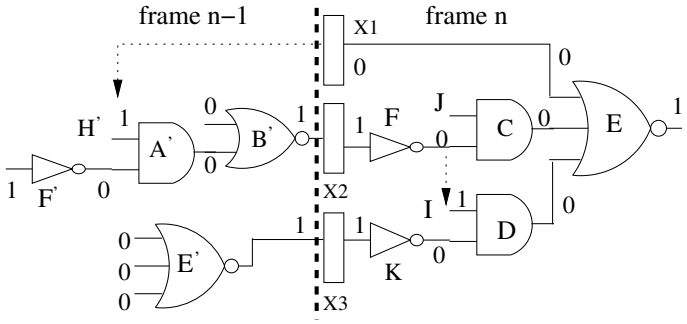


Figure 4.3. Unachievability Learning Example

4.3.4 Unified Framework

In this sub-section, we provide a unified framework which helps us in extracting more illegal states by using the information learned so far. First, for each partial state $F3$ stored in $LIST$ (obtained from the ATPG procedure), we check if it satisfies Theorem 1. If so, $F3$ is marked unachievable and the corresponding clause is learned, which can be used to constrain the induction runs.

Second, we do the following: By performing the ATPG procedure in Section 4.3.1, we obtain groups of flip-flops and impossible combinations among the flip-flops in each group. From the procedure described in Section 4.3.3, we obtain non-trivial implications (from EBL as well as via Theorem 1). For each group of flip-flops obtained at the end of our ATPG analysis, we construct an ordered BDD as shown in Figure 4.4, where each BDD node shows a flip-flop with logic assignments 1 and 0 respectively. At each decision along a path, all the logic implications of the corresponding assignment are injected onto the circuit and logic simulated.

If the new signal assignments due to that decision are not consistent in the circuit, we store the partial-state (so far) as illegal and continue with the opposite assignment for that flip-flop. Otherwise, we check if the partial-state satisfies the criteria of Theorem 1. If so, we store the partial-state as illegal and continue as before. This process is repeated until the whole BDD is enumerated. In Figure 4.4, let the three flip-flops $X1$, $X2$, $X3$ be in one group (in that order) and let $\{X1_0 \wedge X2_1 \wedge X3_0\}$ be an unreachable partial-state learned via our ATPG analysis. As shown in the figure, let the assignment combination $\{X1_1 \wedge X2_1 \wedge X3_0\}$

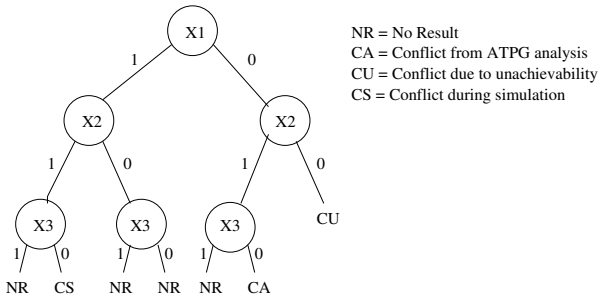


Figure 4.4. Enumeration BDD Example

cause a conflict during logic simulation where as the combination $\{X1_0 \wedge X2_0\}$ satisfies the unachievability criteria. Converting each illegal partial-state to clause form will lead to the three clauses $(X1 \vee \neg X2 \vee X3)$, $(\neg X1 \vee \neg X2 \vee X3)$ and $(X1 \vee X2)$ respectively.

Finally, we simplify all the clauses learned so far. Since we utilized BDD structures, there might be redundancies present in the resulting clauses. For example, from Figure 4.4, we can see that the partial-state $\{X2_1 \wedge X3_0\}$ itself is illegal (irrespective of $X1$). Thus the first two clauses learned above has a redundant literal in them. We utilize binary resolution [19] to remove such redundancies. For example, resolving on the variable $X1$ using those two clauses would lead to the clause $(\neg X2 \vee X3)$ which captures the unreachable core and maximizes the conflicting probability. Similarly, given the ATPG learning from Figure 5.1(b), we would learn the clause ($\neg X1 \vee \neg X3$).

Thus, once we obtain all the constraint clauses, we resolve on each variable at a time and eliminate the redundant literals in them. The final set of clauses maximizes the learned constraints. Recent clause simplifying techniques such as NiVER [30] can as well be used for further simplification, however are not currently employed in our tool.

4.4 Experimental Results

We have integrated all the above learning mechanisms into our framework IFILL (Induction using Fast IL-Legal states). In IFILL, the procedures described in Sections 4.3.1, 4.3.3 and 4.3.4 are performed and the

constraint clauses are obtained. Experiments for sequential circuits from the ISCAS89 and ITC99 benchmark suites were conducted using the zChaff [22] SAT solver (version 2004.11.15 taken from [85]) on a P4 3.2GHz machine with 1GB RAM and running Linux OS. We experimented with the hard safety properties, which are based on aborted state-justifications from a sequential ATPG. An aborted state is a state that the ATPG failed to justify within its resource limit. If s is an aborted state, the property was formulated as $AG(\neg s)$. We have neglected all those properties which can be proved easily using induction.

For the smaller circuits (< 5000 gates), maximum unroll limit was set to 4 and 5 for the ATPG analysis and EBL procedure, respectively, whereas it was set to 3 and 3 for the larger ones. The induction depth k is increased by 25 frames per iteration up to a maximum of 250 frames for the smaller circuits. For the larger ones, these were set to 10 and 100 respectively. Time-out limit for each property was set to 30,000 seconds.

Table 4.1 reports the experimental results. For each instance shown, we report the induction run-times required by the original run (no learning), learning from [74], learning from EBL [75] and with our learning under columns *Orig*, *[74]*, *EBL* and *IFILL* respectively. In column *EBL*, only implications learned via EBL are added, where-as in ours, all the learned illegal states are added in addition. The sub-columns T_p and T_t denote the pre-processing and total times (T_p + solving time) in seconds. As a first observation, we can see that the pre-processing times needed by our method is only few seconds higher than that of EBL. In most cases, they are negligible when compared with the actual induction run-times.

If a property is verified via our technique, we report the corresponding bound in brackets under our sub-column T_t. Among the 26 properties shown, we were able to complete the proof at an early bound for 11 properties whereas the other three approaches could not. For example, for s13207.1, the original run, [74] and EBL took more than 30000, 28000, 10000 seconds respectively without any result, whereas we completed the proof within a total of 129 seconds. Note that, by being able to complete the proofs at smaller depths, the high run-times incurred in solving the higher depth instances are avoided.

For most other properties for which nothing is concluded within the maximum depth, we were able to speed-up the induction runs. For example, for b15850.2, the original run, [74] and EBL took more than 30000, 30000 and 16000 seconds, respectively, whereas we were able to reduce it to 8803 seconds. For the two

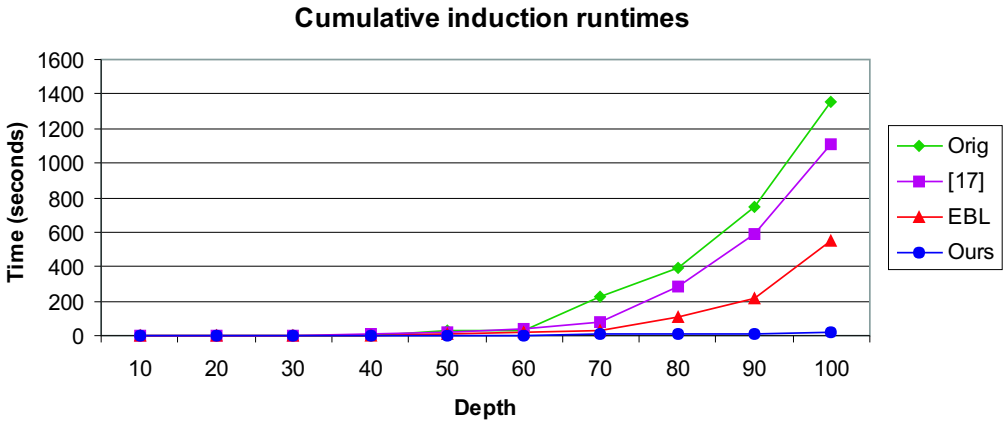


Figure 4.5. Cumulative run-times for b14.2

properties b15.1 and b15.2, adding the new clauses caused negative effects when compared with the original runs. This is because, all the added clauses increase the burden on the SAT engine during BCP and also possibly lead to a different decision ordering, which reduces its efficiency.

Next, we illustrate a cumulative run-time analysis (excluding T_p) for b14.2 with respect to the depth in Figure 4.5. Note that, we want the cumulative increase in run-time with depth to be as slow as possible. For smaller depths, all the runs seemed competitive (differing only little). But when the depth is increased beyond a certain limit (say, 60-70 in this case) the run-times for the other approaches increased almost exponentially, whereas we were still able to contain it. This demonstrates the efficiency of IFILL in enabling a deeper inductive search, under given resource limits.

4.5 Summary

We have proposed novel, low-cost learning techniques to extract illegal partial-states of a design for improving SAT-based induction. These illegal partial-states are replicated as constraint clauses to aid the SAT engine in pruning the search-space. Experiments revealed that our learning can prove many properties at early depths where the conventional runs failed. We were also able to reduce the induction run-times for a

variety of other properties.

There are several directions for future work. First, the above framework aims at *low-cost* and does not guarantee finding all illegal states. For further hard to solve properties, it would be interesting to generalize this learning to capture additional illegal states. Secondly, since only logic implications are employed, the relations might not go deeper. Employing sophisticated techniques such as SAT-based induction itself would be more helpful in extracting the invariants. Finally, the learning regions can be limited for a given property to reduce the pre-processing times further.

Table 4.1. Induction runs for safety properties

Circuit.ϕ	Gates	Orig	[74]	EBL [75]		IFILL(Ours)	
		T_t	T_t	T_p	T_t	T_p	T_t
b10.1	206	83	118	1	8	2	**3**
b10.2	206	86	131	1	7	2	**3**
s526.1	223	3565	3434	1	1291	3	**3[50]**
s526.2	223	4034	4386	1	1599	3	**3[50]**
b13.1	362	534	89	1	16	2	**13**
b07.1	441	193	42	1	11	2	**8**
b07.2	441	219	43	1	20	2	**9**
b04.1	737	580	563	0	**491**	1	509
b04.2	737	768	491	0	**465**	1	507
s1423.1	753	7371	7616	1	8943	4	**7259**
s1423.2	753	7162	9874	1	7475	4	**6013**
b11.1	770	5154	4864	1	3405	2	**2[50]**
b11.2	770	5575	4800	1	2386	2	**2[25]**
b05.1	998	151	46	0	41	1	**1[50]**
b05.2	998	192	32	0	44	1	**1[50]**
s9234.1	5883	28015	21912	18	3033	25	**26[20]**
s9234.2	5883	21319	18682	18	6252	25	**26[20]**
s13207.1	8803	-TO-	28613	108	10712	129	**129[20]**
s13207.2	8803	-TO-	-TO-	108	8258	129	**130[20]**
b15.1	8922	**9037**	11985	490	18120	523	10188
b15.2	8922	**7547**	16851	490	10849	523	8398
b14.1	10098	597	505	48	101	57	**80**
b14.2	10098	1357	1110	48	594	57	**73**
s15850.1	10533	-TO-	-TO-	35	12124	46	**7387**
s15850.2	10533	-TO-	-TO-	35	16051	46	**8803**
s38417.1	23949	10211	7254	34	4787	48	**48[10]**

T_p: Preprocessing time T_t: Total time (with T_p) *TO*: $T_t > 30000$ sec

Chapter 5

Property Strengthening

As seen from the previous chapters, SAT-based BMC/Induction techniques suffer as the depth is increased. In other words, for properties that can be proved only at larger depths, these techniques are not effective. *Strengthening* a property allows it to be falsified/verified at an earlier depth compared to the depth needed to proving the original property. In this chapter, we propose new preprocessing techniques for explicitly identifying co-invariants for a given safety property which are then added to the verification instance for strengthening. First, we employ a path-oriented decision making engine to quickly identify several states which has paths to states violating the property. Next, we generate a set of candidate co-invariants and propose a induction based technique to learn true co-invariants among those candidates. All the learned co-invariants are minimized using resolution and added to the original property to strengthen it.

Below, we first provide our motivation in detail followed by the proposed strengthening techniques and algorithms.

5.1 Motivation

Incremental learning techniques for improving bounded model checking and induction have been exploited in [6, 7, 8]. Since the clause databases over multiple iterations have much similarity, the idea here is to

extract conflict clauses while solving smaller depths and use them to reduce the complexity of solving the larger instances. However, these techniques do not have the capability to reduce the induction depth needed to prove a property when compared with the conventional induction runs. Strengthening a property, on the other hand, involves enlargement of the property such that a counter-example can be seen faster or that it can be verified at a smaller depth. In [86], target enlargement using BDD-based pre-image computations have been explored. However, this technique lacks the ability to learn over various portions of the state-space which affects the final strengthening. Furthermore, since canonical BDDs are needed, this could be memory intensive. In [80], the counter-example obtained during an induction step run is automatically added to the property to strengthen it. Whereas this technique does not require additional run-times, the resulting strengthening power is limited since it largely depends on the counterexample produced by the underlying SAT engine.

Let ϕ be a false property that can be refuted at a depth k from the initial state(s), I. If we are able to extract a set of states S that can reach $\bar{\phi}$ in steps $\leq m$ ($m < k$), then strengthening ϕ with $\neg S$ to form a new property $\phi \wedge \neg S$ can aid in refuting the new property at a depth as low as $k - m$. Similarly, for true properties, strengthening a property will enable it to be proved at an earlier depth, which in turn depends on the quality of the strengthening proposition. This is because as more states are added to the target property, the number of solutions in the induction-step run becomes smaller. For hard to prove properties, it becomes highly beneficial to pre-process it such that it can be proved at smaller depths, thereby avoiding the excessive run-times incurred at the higher depths.

In [80], a property ϕ is strengthened automatically using the counter-example obtained from the induction step run. For example, if s is the first state in the solution for the induction step run, a strengthened property $\phi \wedge \neg s$ is obtained. There are two main limitations in doing so. First, if the induction step run has several such spurious counter-examples (which can be exponential in the number of state-elements), the new property obtained might still require a high induction depth to be proved. Second, the provability of the new property depends on the added strengthening, *i.e.*, adding $\bar{s}_1$ alone might not be sufficient. If we are able to collect several such states, they can be simplified to improve the strengthening. Note that the larger the state-cube added, the better the chances of refuting/proving the resulting property. We exploit the structure of the

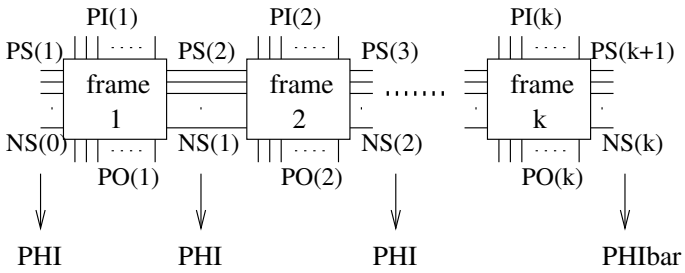


Figure 5.1. Circuit Unrolling

circuit to accomplish this as detailed in the following sections.

5.2 Property Strengthening Using an ATPG

In this section, we propose our ATPG-based framework which aims at explicitly strengthening the property while trying to limit the learning costs. Our main idea is to extract as much information as possible at smaller depths which can be used to improve the quality of strengthening. For simplicity of explanation, we assume a safety property ϕ to be combinational, (*i.e.*, it involves state-elements within a single time-frame) though all our discussions/algorithms can be applied directly to general temporal properties.

We employ a combinational ATPG-based method (using the PODEM algorithm[1] [81]) to identify a subset of states that can violate ϕ. Algorithm 1 outlines our overall procedure. We first unroll a circuit for k time-frames (similar to an Iterative Logic Array expansion) as shown in Figure 5.1, where the unrolled frames are numbered 1, 2, .., k. The lines PI(i), PO(i), PS(i), NS(i) represent the primary inputs, primary outputs, present-state elements, next-state elements for frame i respectively. All the present-state elements for frames 2 to k are simply treated as buffers, while those at PS(1) are treated as pseudo-primary inputs (PPIs).

As shown in Figure 5.1, the present-state elements at frames $1, 2, ..., k$ are constrained with ϕ, whereas the frame $k+1$ is constrained to $\bar{\phi}$. This can be simply done by constructing single-output monitor circuits according to the property clauses and constraining their output to logic 0 or 1. To constrain ϕ at a frame i,

[1]PODEM makes decisions by backtracing from a given objective (say, $gate X = v$) and makes decisions only on the primary inputs and pseudo primary inputs to justify it, (*i.e.*, to set $X = v$)

we need to set the output of the corresponding monitor at frame i to logic 1, whereas for $\bar{\phi}$ it needs to be set to logic 0. This is shown in line 4 of Algorithm 1.

In such a setup, we set a series of objectives $\{\bar{\phi} = 1$ at frame $k+1$, $\phi = 1$ at frames k, $k-1$, ..., $1\}$ and start our decision making process. For the complete algorithm and implementation details of PODEM, the reader is referred to [81]. During the decision making process, we obtain a vector (involving logic assignments to PIs and PPIs) with one of the following three results: (i) that vector sets $\phi = 0$ in at least one of the frames $1, 2, ..., k$, (ii) that vector sets $\bar{\phi} = 0$ at frame $k + 1$, or (iii) that vector satisfies all the set of objectives $\phi = 1$ for frames $1...k$ and $\bar{\phi} = 1$ for frame $k + 1$. Among these three cases, we are interested in the vectors belonging to case (iii) because these are the vectors that satisfy ϕ in time-frames 1 to k, but not in $k + 1$. Once such a vector is obtained during the decision process, we quantify away the PI assignments in that vector since we only need the PPI assignments for strengthening a property. The PPI assignments in that vector form a state-cube which is used to strengthen ϕ. In essence, if $V_{PPI \wedge PI}$ is one such a vector, the new property will be $\phi = \phi \wedge \overline{V}_{PPI}$. Note that the complement of the state-cube V_{PPI}, $\overline{V}_{PPI}$, forms a disjunction of state-element assignments which can be directly added as a clause to the existing property clauses, ϕ. All those vectors belonging to cases (i) or (ii) do not contribute to strengthening and we simply backtrack and continue our decision making process. The above process is shown in line 9 of Algorithm 1.

For efficient search in PODEM, we use the SCOAP measures [83] to identify if a signal in the circuit is most controllable to logic 0 or logic 1. In SCOAP, two controllability parameters $CONT_0$ and $CONT_1$ are computed for each signal, assuming that the primary inputs are fully controllable. In our setup shown in Figure 5.1, we initially set the $CONT_0$, $CONT_1$ of PIs and PPIs to 0 (*i.e.*, fully controllable) and compute the SCOAP values for the rest of the internal signals in the unrolled circuit. These are used during backtracing from the objective to find a suitable PI/PPI logic assignment. To limit the strengthening costs and obtain *bigger* state cubes (*i.e.*, involving a small number of state-elements), we limit the maximum decision level (MDL) in PODEM to a small user-specified value. Note that our main intention is to capture as many state cubes as possible but not to complete the entire search, since a complete search may have exponential complexity.

Algorithm 3 Property Strengthening using PODEM

For ϕ, do base, ind step runs at depth 0; If result, DONE
Given K, $ITER$, MDL; $k = 1$
while ($k \leq K$) **do**
  Unroll circuit k times, set property constraints
  Set $CONT_0$ and $CONT_1$ of PIs, PPIs to 0
  Compute SCOAP values for all internal signals
  $iter = 1$
  while ($iter \leq ITER$) **do**
    Use PODEM to strengthen ϕ : limit decisions to MDL
    Simplify new ϕ using resolution
    Modify $CONT_0$, $CONT_1$ values for PPIs using ϕ
    Recompute SCOAP values for all internal signals
    $iter = iter + 1$
  end **while**
  Do base run at depth k; If failed, DONE; Else, continue
  $k = k + 1$
end **while**

5.2.1 Property Simplification and Reordering

Given a property ϕ, an unroll length k, and a maximum decision limit MDL, the above process yields a strengthened ϕ. However, the state cubes that we obtain are skewed according to the PPIs chosen during the decision process. For example, if a pseudo-primary input PPI_i is chosen at the first decision, then all the state-cubes that we obtain will involve PPI_i. At the same time, for quality strengthening, we need *bigger* state cubes that cover different regions of the state-space. To achieve this, we perform property simplification and input reordering as follows.

First, we employ binary resolution [19] to simplify the strengthened property. Binary resolution is a process in which two impossible assignments can be used to deduce a new impossible assignment. For example, if $(S \vee X_n)$ and $(S \vee \neg X_n)$ are two clauses (where, S is any disjunction of a set of literals), the last variable X_n can be resolved to obtain a newer and stronger clause (S). In line 10 of Algorithm 1, we simplify the strengthened ϕ using resolution. If two clauses $C1$ and $C2$ in new ϕ can be resolved to eliminate a variable, then they are removed from ϕ and the new clause obtained via resolution (say, $C3$) is added to ϕ. We perform the resolution iteratively, wherein each variable in ϕ is resolved to remove any redundant literals in the clauses and the process is repeated until no more resolution is possible. Note that if at least one variable is eliminated via resolution, the resulting ϕ is expected to be stronger than the already strengthened one. Newer simplification techniques such as NiVER [30] could be further used but are not currently employed in our framework. The following theorem ensures the correctness of our approach.

Theorem 5.1: *Property simplification using cross-timeframe resolution is safe, (*i.e., *it does not produce false negatives or false positives during the induction runs).*

Proof: We prove this for the case where the original property ϕ is a single CNF clause and we perform a single resolution, which can be inductively used to prove complex properties or performing multiple resolutions.

Let ϕ consists of a clause $C1 = (S \vee X_n)$, where S is a disjunction of a set of literals and X_n is a state variable. Upon performing our PODEM procedure at an unrolled length k, let $C2 = (S \vee \neg X_n)$ be a new

clause that is added to strengthen ϕ. In other words, $\neg C1$ can be reached in k steps from $\neg C2$. According to our above procedure, $C1$ and $C2$ can be resolved to produce a new clause $C3 = (S)$. We remove $C1$ and $C2$ from ϕ but add $C3$ to it.

First of all, since clause $C3$ consists of a subset of literals of $C1$, it follows directly that (original ϕ is false) $\rightarrow$ (new ϕ is false) and by contrapositivity, (new ϕ is true) $\rightarrow$ (original ϕ is true). The two relations that remain to be proved are (new ϕ is false) $\rightarrow$ (original ϕ is false) and (original ϕ is true) $\rightarrow$ (new ϕ is true).

Let the new ϕ be refuted at a base case depth k' from $I(s)$. In other words, there exists an input vector sequence that can violate the clause $C3$ at depth k'. If that sequence sets state element X_n to logic 0 at depth k', then $C1$ is also violated at depth k'. On the other hand, if that sequence sets X_n to logic 1 at depth k', then $C2$ is violated at depth k' which in turn means that $C1$ will be violated at depth $k' + k$. Thus, we can conclude (new ϕ is false) $\rightarrow$ (original ϕ is false).

Now, let us consider the original ϕ to be a true property which can be verified at an induction step depth k''. In other words, the CNF formula $F1 = \{C1^*(1, 2, ..., k'') \wedge T^*(1, 2, ..., k'' + 1) \wedge \neg C1_{k''+1}\}$ is not satisfiable. Clearly, $k'' > k$, since we already know that $\neg C1$ can be reached in k steps from $\neg C2$. This implies that there is no input sequence in $F1$ that can violate $C2$ at depth $k'' - k$. Due to induction, this in turn follows that there does not exist any input sequence in $F1$ that can violate $C2$ at any depth $> k'' - k$.

Since $C3$ constrains[2] more state space than $C1$, there also does not exist an input sequence that can violate $C2$ at any depth $> k'' - k$ in CNF formula $F2 = \{C3^*(1, 2, ..., k'') \wedge T^*(1, 2, ..., k'' + 1) \wedge \neg C3_{k''+1}\}$. Due to the same reason as above and similar to $F1$, $C1$ cannot be violated at depth $k'' + 1$ in formula $F2$. From the above two arguments, we can conclude that, there does not exist an input sequence that can violate $C3$ at depth $k'' + 1$ in $F2$. In other words, the formula $F2$ is also not satisfiable. Thus, (original ϕ is true) $\rightarrow$ (new ϕ is true). $\diamond$

Via performing resolution on the learned state-cubes, we obtain a further enlarged property ϕ. However, the quality of strengthening can be improved further by capturing cubes from different parts of the state-space. As a consequence, we want to explore the unexplored PPI space of Figure 5.1 (remember that due to the

[2] the number of possible state assignments decreases as the constrained clause is made smaller

MDL limit, we do not exhaust the search). In the following, we use an adaptive procedure to modify the SCOAP values such that backtracing via PODEM leads to a unexplored state-spaces.

Let l_{max} be the maximum clause length in ϕ. Initially, all the $CONT_0$ and $CONT_1$ values for the PPIs are set to 0. For each negative literal L in each clause C of ϕ, we compute $CONT_1(var(L)) = CONT_1(var(L)) + length(C)/l_{max}$, where $var(L)$ is the PPI corresponding to literal L. Similarly, for each positive literal, we compute $CONT_0(var(L)) = CONT_0(var(L)) + length(C)/l_{max}$. The main intuition behind this is to increase the controllabilities of the PPIs in the already found solutions such that newer PPI spaces are favored in the future searches. This is done in line 11 in Algorithm 1.

The SCOAP measures for the entire circuit are recomputed (line 12) and the PODEM procedure is repeated to learn more state-cubes and further strengthen ϕ. The entire process is repeated until a user-defined number of iterations $ITER$ at a depth k. The resulting property ϕ after processing at a depth k is given as the input at depth $k+1$ and the process is repeated until a maximum user-defined depth K.

The overall complexity of the above algorithm is $O(K \times ITER \times 2^{MDL})$, which can be be adjusted according to the user-specified values for K, $ITER$ and MDL.

5.3 Pre-selection and Derivation of Co-invariants

In the previous section, we have presented a procedure to quickly strengthen the property and further improve the quality of strengthening. Its efficiency depends on the decision making process which in turn depends on the accuracy of the SCOAP measures for that circuit. In this section, we present a new technique to identify more co-invariants for a property ϕ via a pre-selection based technique. The main idea here is to generate a set of candidate propositions and check if they are co-invariant with ϕ (*i.e.*, to find if the candidate propositions have to be invariant for ϕ to be invariant). The following theorem forms the basis of our new strengthening algorithm.

Theorem 5.2: *A propositional formula ψ can be used to strengthen ϕ if the following two conditions hold true: (i) the CNF formula, $F = \{\psi^*(1,2,...,k) \wedge \phi^*(1,2,...,k+1) \wedge T^*(1,2,...,k+1) \wedge \neg\psi_{k+1}\}$ is not*

satisfiable and (ii) no input sequence can violate ψ within depth k from $I(s)$.

Proof: First, consider ψ to be an invariant. Because strengthening a property with an invariant cannot cause false negatives or false positives, strengthening ϕ with ψ will always be sound.

Next, consider ψ not to be an invariant, *i.e*, let ψ be refuted at a depth k' from $I(s)$ via a path $P = s_1, s_2, s_3,$ ..., $s_{k'+1}$, where $s_1 \in I(s)$. Clearly, since the theorem states that no input sequence violates ψ within a depth of k from $I(s)$, $k' \geq k$. Next, consider a CNF formula $F1 = \{\psi^*(1, 2, ..., k) \wedge T^*(1, 2, ..., k+1) \wedge \neg\psi_{k+1}\}$. Since ψ is not an invariant, $F1$ will be satisfiable. Now, if F is unsatisfiable and $F1$ is satisfiable, we can conclude that at least one of the final $k+1$ states in path P (which has a length $k'+1$) must violate phi. Since such a state starts from $s1 \in I(s)$, we can also conclude that phi is not an invariant, thereby reducing the induction depth. $\diamond$

If the two conditions of Theorem 2 are satisfied, then ϕ can be strengthened using ψ to form $\phi \wedge \psi$. In order to efficiently utilize this, we first generate a set of propositions ψ as given in line 1 through 8 of Algorithm 2. Essentially, we order the state elements of the design according to their structural connectivity using the MLP[3] procedure [84]. All the combinations among groups of M adjacent state elements are enumerated whose disjunctions are treated as suitable candidates. For example, if X_1, X_2 and X_3 are 3 adjacent state-elements in the order, we obtain ($X_1 = 0 \vee X_2 = 0 \vee X_3 = 0$), ($X_1 = 0 \vee X_2 = 0 \vee X_3 = 1$), ..., ($X_1 = 1 \vee X_2 = 1 \vee X_3 = 1$) as the 8 potential co-invariants. Given a depth k, we initially check each candidate if they can be reached from the initial state(s). If so, that candidate is removed and not considered anymore. On the other hand, if refutation is seen, we check that candidate for co-invariancy using Theorem 2. As in Algorithm 1, we use a maximum decision limit, MDL, in the PODEM algorithm to limit the amount of time spent in identifying the co-invariants. If at least one solution is found for F (see Theorem 2 statement), then we can conclude the corresponding ψ is not a co-invariant and abort the search. On the other hand, if no solution is found, we can conclude ψ to be a co-invariant and use it to strengthen ϕ (line 15). The resulting ϕ at the end of depth k can be further simplified using resolution (line 17) which is then used as the input property at depth $k+1$. This process is repeated until the user-defined maximum depth K is reached.

[3]MLP orders state elements according to their input support affinity

Algorithm 4 Property Strengthening using Pre-selection

Given Max candidate size, M
Order the state elements using MLP
for each element X_i in ascending order **do**
  $X_{i+1}, X_{i+2}, .., X_{i+M-1}$ = Next $M-1$ elements in order
  Enumerate all 2^M combinations among $X_i, X_{i+1}, .., X_{i+M-1}$
  Add disjunction of each enumeration to CAND-LIST
  Remove X_i from the order
end for
Given K, MDL; $k = 1$
while ($k \leq K$) **do**
  for each $\psi \in$ CAND-LIST **do**
    If base run at depth k failed, remove ψ; Else continue
    Unroll circuit k times
    Check Theorem 2 using PODEM : limit decisions to MDL
    If Theorem 2 holds, set $\phi = \phi \wedge \psi$ and remove ψ
  end **for**
  Simplify ϕ using resolution
  $k = k + 1$
end **for**

5.4 Overall Flow and Discussion

In our current implementation, we combined both algorithms such that the strengthened property can be alternated. For a given unrolled depth k (k = 1, 2, ..., K), we first run Algorithm 1 and use the obtained strengthened property as input to Algorithm 2. The strengthened property from Algorithm 2 is again used as input at depth $k+1$ for Algorithm 1, and so on. After reaching the maximum user-defined depth K, the resulting strengthened property is checked using conventional SAT-based induction.

For true safety properties, a successful induction run is sufficient. In other words, the depth at which they are proven is often immaterial to the designer. However, for false properties, it is important to generate the counter-example that refutes the property so that the design can be debugged accordingly. Note that, when we strengthen a false property, it is guaranteed to be refuted at a depth k' ($\leq k$), where k is the depth at which the original property could be refuted. Whereas the new clauses added to strengthen the property might increase the intuition on the part of the designer, the shorter counter-example might not help in tracing the cause for the original bug. Since we strengthen a property by searching the state transitions and simplifying them using resolution, the original counter-example can be reconstructed completely from the new counter-example obtained. Since we know the cause for adding a new clause C to strengthen ϕ, a counter-example for C can be appended with the input vector sequence that caused $\neg\phi$ to be reached from $\neg C$. This can be automated according to the proofs of Theorems 1 and 2.

5.5 Experimental Results

We have developed a prototype tool for explicit property strengthening, which takes a safety property as input and produces the strengthened property. Induction runs were conducted using a general purpose SAT solver zChaff [85] and a special purpose incremental solver for induction called Temporal Induction Prover (TIP) [7]. The main motivation is to evaluate the proposed property strengthening with respect to both the general purpose learning mechanisms used in the modern SAT solvers and the state-of-the-art incremental learning techniques for induction.

All the experiments were conducted on a 3.2GHz Pentium 4 machine running Linux OS with 1GB of RAM.

We experimented with the safety properties for the ISCAS89 and ITC99 benchmarks which are properties representing aborted states from a Sequential ATPG. If s is an aborted state, AG($\neg s$) is formulated as a safety property. For each property, we ran zChaff and TIP until a maximum induction depth of 250 or until a time out limit of 7200 seconds. For all the circuits, we have set the maximum unroll depth K to 3, the maximum decision limit, MDL to 20 and the number of iterations, $ITER$ to 3. In Algorithm 2, M is set to 3.

Table 5.1 shows the experimental results. For each circuit, *FFs* and *Gates* show the number of flip-flops and gates respectively. The results for the properties with the conventional runs are shown under the column *Original*, while those with the proposed techniques are shown under *Ours*. The columns termed *Time* gives the induction run-times in seconds whereas the column *Time(str)* reports the preprocessing runtime for strengthening. The column *#Cls(str)* reports the number of strengthening clauses learned for the respective property. The columns *Result* report the final result where F means "failed", P means "passed" and N means "nothing concluded". The number next indicates the depth at which the result is obtained or the maximum depth solved until the time limit is reached. Among all the techniques, we highlight the lowest run-times and the lowest depths for those proven properties in **bold**.

First, consider our processing run-times needed to strengthen the property. These depend mainly on the size and structure of the circuit and varies according to the target property. The maximum pre-processing time taken is around 57 seconds for the circuit s15850.1. Note that in most of the cases, these times are insignificant when compared to the actual induction run-times. Similarly, the number of strengthening clauses depends on the complexity of the circuit and the number of total solutions present for that property. A maximum of 426 clauses were learned for the instance s444.1.

Next, consider the actual property checking results. Depending on the performance of our technique, we classify the results into 3 categories. The first category are those where our property strengthening was able to improve the total run-times required to falsify/verify the property. This is mainly due to its ability in reducing the depths required to produce a result. As an example of a false property, consider s1512.1. When the original property is checked using zChaff and TIP, they both timed out while reaching depths of 123 and 188 respectively. On the other hand, the strengthened property was falsified at a depth of only 2 time-frames.

Note that if resolution is not performed but the learned state-cubes are simply added for strengthening, the improvement in the proving depth for false properties can only be linear. For example, if N is the depth needed to refute a property using the conventional run and we learn state-cubes up to a depth K, then the lowest depth at which the enlarged property can be refuted is only $N - K$. Since we employ resolution to improve the quality of strengthening, we were able to reduce the proving depths significantly for the proven false properties (such as s1512.1, b11.2, s15850.1 *etc*). Next, consider the passing properties. For example, for s526.2, zChaff timed-out after depth 176 while TIP took 888 seconds until the maximum depth of 250 without any conclusion. On the other hand, the strengthened property was proved at a depth of 107 with 31 and 22 seconds with zChaff and TIP respectively. In several cases, due to the reductions in the proving depths required, the run-times have been improved by more than an order of magnitude. Also, note that the improvements achieved over the general purpose solver zChaff is in general more when compared with the prover TIP, since TIP already obtains performance improvement due to its incremental solving capability.

Next, consider the second category where we were able to reduce the induction depths but could not improve the run-times compared to TIP. For example, for s444.1, the original property was proved false at a depth of 139 in 53.8 seconds by TIP. Although the strengthened property was proved false at a depth of 84, the runtime needed was around 93.5 seconds. This is because, all the clauses that are added to strengthen the property can become a burden on the SAT engine during BCP. At the same time, they skew the number of clauses each variable appears in, which possibly leads to a different variable order chosen by the SAT solver, thereby reducing its efficiency. For the passing property b05.1, our total runtime was higher due to the pre-processing time, though the actual proving time is negligible. In all these cases, however, we were still able to obtain considerable improvement over zChaff.

Finally, consider the last category, where our technique could not reduce the induction depths. Most of these properties are hard and could not be proved within the maximum depth of 250 or until the timeout limit. In these cases, again, the new clauses added for property strengthening incur additional computational costs and hence the depth reached within the time limit is reduced.

5.6 Summary

In this chapter, we proposed new and low-cost techniques for explicitly strengthening a safety property such that it can be proved at earlier induction depths. Experiments revealed that our techniques can be robust in complementing the incremental learning techniques for induction and improve the run-times for both failing and passing properties. We used PODEM as a basis of our implementation, although our algorithms can be used in conjunction with the newer solvers such as the hybrid SAT solver proposed in [87].

There are two important directions for future work. First, it has been observed that property strengthening can sometimes be a overhead due to the additional clauses added. Clever minimization of the added clauses would be interesting to pursue. For example, some of the added clauses might not present any value for strengthening when certain other clauses are added. Secondly, it will be interesting to exploit ways of automatically identifying co-invariants for a given safety property (using Theorem 5.2) as opposed to pre-selecting them as done in our work.

Table 5.1. Results for induction runs for safety properties

Circuit.ϕ	FFs	Gates	Original				OURS					
			zChaff [85]		TIP [7]		Time	#Cls	zChaff [85]		TIP [7]	
			Time	Result	Time	Result	(str)	(str)	Time	Result	Time	Result
b10	66	189	1876	N:250	-TO-	N:23	1.4	3	**1.4**	**P:1**	**1.4**	**P:1**
s382.1	21	266	71.1	P:100	1.6	P:100	0.5	144	**0.5**	**P:1**	**0.5**	**P:1**
s400.1	21	276	11.6	P:61	0.7	P:61	0.6	174	**0.6**	**P:1**	**0.6**	**P:1**
b13.1	53	362	70.1	F:75	29.2	F:75	1.1	103	48.5	**F:73**	**26.2**	**F:73**
b13.2	53	362	-TO-	N:212	898	N:250	1.0	95	-TO-	N:218	**575**	N:250
s526.1	21	368	-TO-	N:174	209	P:187	1.7	175	59	**P:127**	**41**	**P:127**
s526.2	21	368	-TO-	N:176	888	N:250	1.9	161	31	**P:107**	**22**	**P:107**
b11.1	30	396	-TO-	N:115	-TO-	N:121	2.1	142	**2.1**	**P:1**	**2.1**	**P:1**
b11.2	30	396	1801	F:73	228	F:73	2.3	168	**75**	**F:38**	188	**F:38**
b07	51	433	-TO-	N:192	4018	N:250	3.2	29	4476	N:250	**3372**	N:250
s641	19	510	20.5	N:250	-TO-	N:149	2.4	45	**2.4**	**P:1**	**2.4**	**P:1**
s713	19	529	22.1	N:250	-TO-	N:158	2.5	43	**2.5**	**P:1**	**2.5**	**P:1**
b04.1	66	546	-TO-	N:188	3984	N:250	2.3	28	6991	N:250	**953**	N:250
b04.2	66	546	-TO-	N:202	3301	N:250	2.3	29	-TO-	N:247	**1502**	N:250
s1423	74	779	-TO-	N:86	-TO-	N:138	3.5	122	3.8	**F:7**	**3.6**	**F:7**
s1512.1	55	967	-TO-	N:123	-TO-	N:188	3.7	33	**3.7**	**F:2**	**3.7**	**F:2**
s9234	170	6331	-TO-	N:58	-TO-	N:82	36	216	**36**	**F:1**	**36**	**F:1**
s15850.1	506	11055	1437	F:32	293	F:32	57	191	**57**	**F:1**	**57**	**F:1**
s15850.2	506	11055	558	F:27	145	F:27	53	167	**53**	**F:1**	**53**	**F:1**
s382.2	21	266	129	F:91	**7.1**	F:91	0.6	137	24.2	**F:77**	10.4	**F:77**
s400.2	21	276	133	F:91	**6.9**	F:91	0.6	78	26.1	**F:68**	9.9	**F:68**
s444.1	21	293	1133	F:139	**53.8**	F:139	1.1	426	61	**F:58**	92.3	**F:58**
s444.2	21	293	16.7	F:56	**2.1**	F:56	1.2	225	2.7	**F:25**	4.2	**F:25**
b05.1	34	642	9.8	P:58	**0.9**	P:58	2.4	16	2.4	**P:3**	2.4	**P:3**
b05.2	34	642	6.7	P:31	**0.7**	P:31	2.6	17	6.1	**P:30**	4.5	**P:30**
s1512.2	55	967	-TO-	N:134	-TO-	N:180	3.9	72	-TO-	N:119	-TO-	N:164
s3384	183	1937	-TO-	N:64	-TO-	N:51	6.2	30	-TO-	N:58	-TO-	N:42
s13207	550	9517	-TO-	N:48	-TO-	N:86	47	12	-TO-	N:45	-TO-	N:73

F: Failed P: Passed N: Not known; Next # shown is the depth reached (max=250); -TO-: Time Out after 7200 seconds

Chapter 6

Interleaving BMC and Bounded Reachability Analysis

In this chapter, we present a novel SAT-based framework that has the potential to search exponentially more new states than those traversed via conventional approaches. Our technique employs forward and backward bounded reachability analysis together with a bounded model checker to efficiently traverse the state-space. The most important feature of our technique is the controllability-analysis-based guidance that utilizes the knowledge from the previous searches to direct the current search in order to reach the target state(s) faster. In the following sections, we first present our motivation followed by the proposed framework and experimental results.

6.1 Motivation

As mentioned in chapter 2, the efficiency of induction techniques depends mainly on the underlying SAT solver employed. As the induction depth is increased, the number of variables and clauses increase linearly. This in turn increases the search complexity exponentially, and it may become unmanageable for the SAT engines. Due to practical resource limitations, this bound is often restricted to a small value depending on

the size of the design. Thus, all the reachable states that lie beyond this bound would not be traversed via conventional bounded model checkers.

The sequential SAT solver proposed in [88] tries to avoid this problem by doing a prioritized search backward from the target state. Although the presented heuristics try to maximize the solution sizes, the number of states they traverse might still be limited because they obtain few solutions at a time and check if the solution spaces contain the initial state. Furthermore, if the backward search reaches a very long path in the illegal state space, all the solutions that were computed would be futile.

To overcome the above problems, we combine depth-first forward and backward bounded reachability analysis together with k-induction. We use induction in order to make our framework efficient for passing properties; however, the proposed algorithm can be restricted by the user to just the base case runs if obtaining a counter-example is the primary goal.

6.2 Proposed Searching Framework

In this section, we describe the proposed framework where we augment bounded reachability analysis together with bounded model checking. The basic idea is given in the next subsection followed by our controllability based guiding heuristic.

6.2.1 Basic Idea

The number of reachable states in a design could be exponential in the number of state-elements. Suppose we can extract a set of states, S_k, from the state-transition graph such that any state $\notin S_k$ can be reached from at least one state $\in S_k$ within a distance k. In other words, the shortest distance between a state $\notin S_k$ and a state $\in S_k$ is at most k. When $k = 0$, the set S_0 has all reachable states in it and is unique. On the other hand, for $k \geq 1$, there can be more than one such sets that can be extracted from the state transition graph. Let L_k represent the minimal-sized set possible for a given k. As the distance k is increased, L_k either remains the same or decreases. For example, consider the state transition graph shown in figure 6.1(a). There are 8

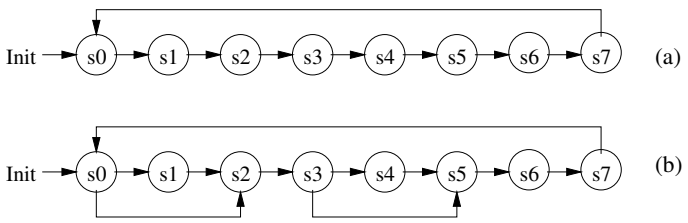


Figure 6.1. Example STGs

reachable states and hence $L_0 = 8$. Further, since each state has at most one out-going edge, S_1 could be one of $\{s_0, s_2, s_4, s_6\}$ or $\{s_1, s_3, s_5, s_7\}$. Hence, $L_1 = 4$. Similarly, $L_2 = L_3 = 3$, $L_4 = L_5 = L_6 = 2$ and $L_7 = 1$. Now, consider the STG in Figure 6.1(b), where two new edges s0 $\rightarrow$ s2 and s3 $\rightarrow$ s5 are added. Now, S_1 can just be $\{s_0, s_3, s_6\}$, thus making $L_1 = 3$ in this new graph. Consequently, as more edges are present in an STG, the corresponding minimal set sizes can be smaller. In a hardware design, the number of states reachable in one step from a given state s can be exponential in the number of primary inputs. As a result, the reduction in L_k could be exponential as k is increased.

Clearly, if we can extract one such set for an appropriate k, we can employ bounded model checking by setting the initial states to all those in S_k and search within a maximum bound of k. If a target T was not found within k distance from any state in S_k, then we can conclude that T is not present in the STG, i.e., it is unreachable. The value of k poses a trade-off in such a scenario. If k is small, the maximum bound needed for verification will be small but the size of S_k might be huge to compute. On the other-hand, if k is made larger, the set S_k might be smaller but the maximum depth needed for verification might be higher. However, the main bottleneck here would be the computation of S_k since guaranteeing that all states outside S_k are within k distance is itself another reachability problem. As a result, an exact computation of S_k would be impractical even for medium sized designs.

In our approach, we try to approximate S_k such that the computation time is less intensive. The main idea behind our approach is shown in Figure 6.2 where three unrolled circuit blocks are shown. The first one represents the forward reachability analysis engine, the middle one represents a bounded model checker and the last one represents the backward reachability analysis engine. First, consider the forward reachability engine where each circuit blocks be numbered 0 to D_f. Let FWD represent the set of initial states we have

at any given time. We constrain the 0^{th} state-space to FWD and the remaining 1 to D_f to $\neg FWD$. Under such constraints, we run the SAT engine to obtain a satisfying solution. If a solution indeed exists, then the satisfying assignments for state elements at blocks 1 to D_f form new reachable states $s_1, s_2, ..., s_{D_f}$ which can be added to FWD and the process can be repeated.

Since the bounded model checker performs breadth-first search, we obtain solutions in the forward analysis engine in a depth-first manner. Initially, we have $FWD = \{I\}$, the original initial (or reset) state. After obtaining one solution, we have $FWD = \{I, s_1, s_2, ..., s_{D_f}\}$. When computing the next solution, we always try to satisfy s_{D_f} first in the 0^{th} frame. If no solution exists, we continue to satisfy s_{D_f-1} and so on. At a given time, if there does not exist a solution starting from any state $\in FWD$, then we have reached a *bounded* fixed point. In other words, it would be sufficient to search for the target T within a depth D_f from states $\in FWD$ using the bounded model checker.

By using the depth-first search in this manner, we obtain a state-sequence in the state transition graph. The main intuition here is that, by constraining the state-elements at frames 1 through D_f to $\neg FWD$, the SAT solver implicitly eliminates all states that can be reached from those states $\in FWD$ within a distance $D_f - l$ and can *only* reach those states $\in FWD$ within a distance l, where $1 \leq l \leq D_f$. Such states can be exponential in number and would be covered by the bounded model checker when FWD is used as the initial state set. Thus, the new states obtained in a solution would add value to the existing set FWD. There are two main points to be noted here: (i) As more depth-first states are added to the set FWD, the number of states rejected by the SAT engine can increase exponentially and (ii) the effectiveness of resulting states in FWD depends on the depth D_f. Clearly, larger D_f would be beneficial in rejecting more states but at the cost of higher computation times.

Likewise, the backward reachability analysis can be formulated in a similar fashion. Initially, the set of target states, BWD is set to $\{T\}$ which is the first target state. We constrain the state-space at frame D_b to BWD (refer to Figure 6.2) while those at frames 0 through $D_b - 1$ to $\neg BWD$. Once we obtain a solution, the resulting states are added to BWD to become $\{T, s_{D_b-1}, ..., s_1, s_0\}$. The process can be repeated and more states are added to BWD.

Once a given number of states have been computed to FWD and BWD, we employ the bounded model checker to verify if any state $\in BWD$ can be reached from any state $\in FWD$ within a bound D_{bmc}. This framework can potentially be very powerful because the sets FWD and BWD approximate the previously explained S_k and by using a bounded model checker, we are verifying exponential number of states at one time. However, we have observed that repeating the bounded model checking for depth D_{bmc} after every iteration could be time consuming. So, initially we start with $D_{bmc} = 1$, and then gradually increase this bound as more states are added to FWD and BWD.

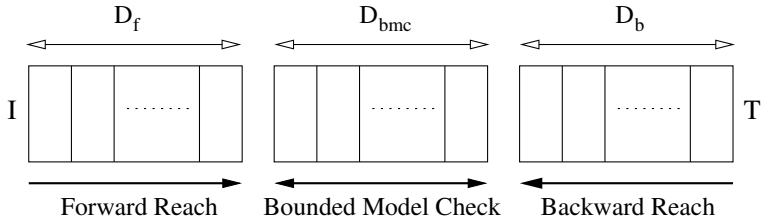


Figure 6.2. Basic idea of proposed method

6.2.2 Controllability-based Guiding

In the previous subsection, we have presented a basic framework for computing FWD and BWD. However, the states in FWD and BWD are computed independently and the new solutions obtained largely depend on the decision order followed by the underlying SAT solver. In this work, we try to guide the computation of FWD and BWD using controllability measures such that the target can be reached faster. In other words, we compute FWD keeping in mind those states that cannot lead to states $\in BWD$. Similarly, BWD is computed by avoiding states that cannot be reached from FWD. The main idea is based on our observation that certain state assignments are hard to be justified from the initial state and that the target state is hard to be reached from certain state assignments. For example, some state elements may remain in the same initial state value within a certain bound. If the target state requires the opposite values on those elements, it would be difficult for conventional bounded model checkers because they might need to unroll deeper.

Initially, we compute SCOAP measures [83] for the circuit. Each internal signal in the circuit is initially

associated with very large forward controllability index values for logic 0 and logic 1. Since the primary inputs are fully-controllable, their indices are set to 0. These indices are propagated onto the circuit to obtain new controllability values. This process is repeated until all values converge in the forward direction. Let M_0 and M_1 be the maximum forward controllability indices for any state-element. We identify all those that has an index greater than a threshold (say, 70%) of the maximum values to be hard to control in the forward direction. All these assignments are stored as a list $FWD-Cont$. Similarly, we set large backward controllability values for all signals by setting the backward controllability for primary outputs to be 0. The values are propagated backward until they converge. Again, we identify a set of state elements that are hard to control in the backward direction. All these assignments are stored in a list $BWD - Cont$.

During our forward bounded reachability computation, we assert each of the hard-to-control assignments $\in FWD - Cont$ at frame D_f and try to obtain a solution. If one exists, we add the solution states to FWD and the corresponding assignment is removed from $FWD-Cont$. On the other hand, if no solution exists, we mark the corresponding state-variable assignment as truly hard in the forward direction. When performing backward bounded reachability computation, we assert the opposite values for these marked state assignments at frame 0 such that the solutions obtained for BWD will be biased toward those in FWD.

Likewise, during backward bounded reachability computation, we assert each of the hard to control assignments $\in BWD - Cont$ at frame 0 and try to obtain a solution. If one exists, we add the solution states to BWD and the corresponding assignment is removed from $BWD-Cont$. On the other hand, if no solution exists, we mark the corresponding state-variable assignment as truly hard in the backward direction. When performing forward bounded reachability computation, we assert the opposite values for these marked state assignments at frame D_f such that the solutions obtained for FWD will be biased toward those in BWD.

Thus, our overall flow is as follows: Initially, we obtain the two sets $FWD - Cont$ and $BWD - Cont$. At the start of every iteration, we check these for controllability and identify those truly hard ones. Then we perform forward and backward bounded reachability computations followed by bounded model checking with induction. If a result is not obtained, then we goto the next iteration where we try to justify the still remaining hard to control assignments. If a solution is found for a state assignment, instead of simply

removing it from the control lists, we also form new candidates by conjoining those for which a solution has already been found. This way, we enhance the sets $FWD-Cont$ and $BWD-Cont$ with multi-literal state assignments. For example, let $F1 = 0$ and $F2 = 1$ be our initial candidates in $FWD - Cont$. Once we find a solution for both of these, we remove these from the list and obtain a new candidate $F1 = 0 \wedge F2 = 1$ and add it to $FWD - Cont$. To avoid excessive overheads, we limit each candidate size to 4.

6.3 Overall Algorithm

Our overall algorithm for augmenting bounded reachability analysis with bounded model checking is shown in Algorithm 5, where we assume the target state(s) s_t to be those satisfying the property ϕ. During forward reachability, we obtain N number of solutions per iteration and store it in the list FWD. Similarly, in backward reachability, we obtain N number of solutions per iteration and store it in the list BWD. After each iteration, we use our BMC checker to see if any state in BWD can be reached from any state in FWD. If so, we can conclude that ϕ is not invariant. Otherwise, we do the induction step run using BWD. If this run is UNSAT, then we conclude that ϕ is invariant. If nothing is concluded, then we perform next iteration by computing and adding more states to FWD and BWD.

6.4 Experimental Results

All the proposed techniques are built on top of the zChaff SAT solver [85] (version 2004.11.15). We compare the performance of our techniques with the conventional induction runs and with two other existing techniques TIP [7] and Sequential SAT solver [88]. TIP performs induction runs using incremental learning and other optimizations for specifying unique-state constraints on the fly. Experiments are conducted for justifying partial-states that are aborted after 10,000 seconds during test generation. We check the efficiency of the SAT-based methods in solving these hard problems. In our technique, the bound for forward/backward reachability is set to 5 and the number of solutions obtained per iteration is set to 100.

Table 6.1 shows the state justification results with a time-out limit of 3600 seconds set for each run. For each

Algorithm 5 Interleaving Bounded Reachability with Bounded Model Checking

Given initial state s_0 and property ϕ. Let target state(s) s_t satisfies $\bar{\phi}$
Given depths $D_f = D_b = k$ and number of solutions N; set $D_{bmc} = 1$
set $FWD = s_0$ and $BWD = s_t$
while (not DONE) do
    ObtainFWD(D_f, N)
    ObtainBWD(D_b, N)
    DoBMC(D_{bmc}, FWD, BWD)
    If result seen, then DONE
    $D_{bmc} = D_{bmc} + 1$
end while
ObtainFWD(D_f, N) {
    numSolutions = 0
    while (numSolutions $< N$) do
        Constrain T_0 to last state in FWD
        Constrain $T^*(1, 2, ..., D_f)$ to $\neg FWD$
        SAT-Check(ForwardReach)
        Append solution states (s_1, s_2, ..., s_{D_f}) to FWD
        numSolutions = numSolutions + 1
    end while
}
ObtainBWD(D_b, N) {
    numSolutions = 0
    while (numSolutions $< N$) do
        Constrain T_{D_b} to last state in BWD
        Constrain $T^*(0, 1, ..., D_b - 1)$ to $\neg BWD$
        SAT-Check(BackwardReach)
        Append solution states (s_{D_b-1}, s_{D_b-2}, ..., s_0) to BWD
        numSolutions = numSolutions + 1
    end while
}
DoBMC(D_bmc, FWD, BWD) {
    set $s_0 = FWD$ and $\phi = \neg BWD$
    SAT-Check(BaseCase)
    If SAT, counter-example found
    Else SAT-Check(IndStep)
        If UNSAT, property is true
        Else, nothing is concluded
}

instance, we first show the number of primary inputs, flip-flops, and the number of gates in columns 2, 3 and 4, respectively. The column *State Result* shows the nature of the corresponding state, where *sat/unsat* mean that the state is reachable/unreachable, respectively. The status *unknown* means that no technique could classify that state within the time-limit. Each of the remaining of the columns first report the runtime for the corresponding technique, and then report in brackets the counter-example length for satisfying properties for the Conventional Induction, TIP, and SeqSAT, respectively. Whenever any of the induction based techniques timed out (TO), the number in brackets indicate the depth reached. Under *OURS* columns, in addition to the execution time, we report in brackets (i) the counter-example length, (ii) induction depth at which the result is obtained, and (iii) the iteration number at which we obtained the result. Take instance s1423.2, for example. This is a second aborted state for s1423, which has 17 primary inputs, 74 flip-flops, and 779 gates. The conventional induction method timed out at depth 110, TIP timed out at depth 142, SeqSAT also timed out. Our base-line method timed out at iteration 16. On the other hand, with controllability based guiding, our technique was able to reach the target state with a length of 6823 time-frames, which was obtained at induction depth of 4 at iteration #13. Out of the 14 instances, our techniques were able to classify 10. On the other hand, the three other techniques could not finish for many of these instances.

Consider the instance b15.1. Though the target was reached within a depth of 14 via regular BMC, our base-line took 1520 seconds while the controllability based guding timed out. This clearly indicates that our technique is not an alternative to BMC but can complement it well.

In Table 6.2, we report additional analysis regarding the reachability of the states obtained by our technique. We obtain 1000 forward solutions using a bound of 5 (i.e., a total of 5000 unique states) and contrast this with 5000 unique states obtained using random simulation. For example, consider s1423. The first 100 unique states obtained by the random simulation can be reached from the initial state within a depth of 10. This is intuitive since the random simulation starts from the initial state, and the states reached would not be very far. Likewise, in our technique, the first 100 unique states obtained are also not too far from the initial state. 91 (81+10) of these 100 states obtained by our technique are within a distance of 10 from the initial state while only 9 are beyond a distance 10. When the last 100 are checked, we can see that most of those obtained by random simulation can still be reached within a distance of 10 from the initial state. On

Table 6.1. State Justification Experiments on ISCAS and ITC Benchmarks

Instance	PIs	FFs	Gates	State Result	Conv Ind	TIP [7]	SeqSAT [88]	OURS	
								Base-line	With Cont
s1423.1	17	74	779	sat	1(5)	1(5)	TO	3(108, 2, 0)	3(108, 2, 0)
s1423.2	17	74	779	sat	TO(110)	TO(142)	TO	TO(-, -, 16)	2369(6823, 13, 4)
s3384	43	183	1937	unkwn	TO(34)	TO(41)	TO	TO(-, -, 26)	TO(-, -, 24)
s9234	77	170	6331	sat	TO(51)	TO(73)	319(2811)	TO(-, -, 35)	TO(-, -, 30)
s13207.1	62	638	9517	unkwn	TO(43)	TO(55)	TO	TO(-, -, 23)	TO(-, -, 21)
s13207.2	62	638	9517	sat	TO(51)	TO(64)	TO	TO(-, -, 24)	177(1033, 2, 1)
s15850.1	77	534	11055	sat	TO(45)	TO(53)	TO	TO(-, -, 23)	778(4675, 6, 2)
s15850.2	77	534	11055	unkwn	TO(51)	TO(65)	TO	TO(-, -, 24)	TO(-, -, 20)
s35932	35	1728	18148	sat	161(8)	88(8)	TO	106(532, 2, 0)	106(532, 2, 0)
s38584	38	1426	23706	sat	182(12)	101(12)	TO	88(698, 2, 1)	88(698, 2, 1)
b07*	2	51	433	unsat	TO(192)	TO(387)	59	1(-, -, 3)	2(-, -, 3)
b12	6	121	1119	sat	206(29)	145(29)	TO	368(4324, 8, 3)	288(3726, 7, 3)
b15.1	36	447	7378	sat	638(14)	239(14)	TO	1520(8073, 15, 3)	TO(-, -, 19)
b15.2	36	447	7378	sat	TO(25)	TO(30)	TO	412(3236, 8, 2)	1221(5723, 10, 2)

All times are in seconds. TO: Runtime exceeded 3600 seconds. *Ours reached bounded fixed point at iteration 3

the other hand, by our technique, the last 100 are significantly farther from the initial state. For s1423, all of these last 100 states are beyond the distance of 30 from the initial state. Note that the actual distance can be much longer, but we did not verify exhaustively due to exceeding runtimes required. Similarly, for the larger circuits s35932 and b15, the actual distances could be much greater than 10.

Table 6.3 shows the analysis for maximum induction depth reached as more reachable states are learned and added to the initial states. In the experiments, the bound for reachability analysis is again set to 5, and a given number of states are learned using our base-line forward analysis which are then used to perform the induction runs. For the number of states given in each column, we report the corresponding runtime required to compute these states followed by the maximum bound reached within a time limit of 3600 seconds. There are two main observations: (i) It is clear that the time required to compute new solutions increases super-linearly. For example, in instance s1423.2, it took 1 second to compute 1000 unique states,

Table 6.2. Reachability of First/Last 100 States among 5000 states

Circuit	Type	States	Distance from Init State			
			0-4	5-9	10-29	≥30
s1423	Rand	First 100	86	14	-	-
	Ours	First 100	81	10	9	-
	Rand	Last 100	96	4	-	-
	Ours	Last 100	-	-	-	100
s35932	Rand	First 100	99	1	-	-
	Ours	First 100	4	5	91 (≥10)	
	Rand	Last 100	99	1	-	-
	Ours	Last 100	-	-	100 (≥10)	
b12	Rand	First 100	100	-	-	-
	Ours	First 100	23	36	41	-
	Rand	Last 100	98	2	-	-
	Ours	Last 100	-	19	81	-
b15	Rand	First 100	99	1	-	-
	Ours	First 100	13	5	82 (≥10)	
	Rand	Last 100	100	-	-	-
	Ours	Last 100	-	-	100 (≥10)	

2 seconds to compute 2000 unique states, but it took 18 seconds to obtain 10000 unique states. This is because, as more states are used to constrain the state-spaces, the SAT solver takes more time to search for new solutions. (ii) The maximum induction depth reached varied insignificantly as more initial states are injected. For example, in instance 1423.2, with 1000 initial states, we were able to reach the depth of 109 time-frames within the time-limit. When we start with 10000 initial states, we were able to reach a depth of 101 time-frames. This is because, most of the runtime is consumed at the higher induction depths, and adding more initial states caused not reaching only a few of these higher depth instances. This shows that our technique can also be used to statically obtain a certain set of states via forward/backward analysis, and then conventional induction can be used for furthering the proof. One such case is s15850.1, where after the first 1000 solutions were obtained, the target was reached at a bound of 25.

Table 6.3. Maximum induction depth reached in one hour

Instance	#InitStates					
	1000	2000	3000	4000	5000	10000
s1423.2	1:109	2:105	3:105	4:105	6:104	18:101
s3384	1:33	4:33	7:33	11:32	17:32	55:31
s9234	5:50	11:49	18:49	25:48	33:48	83:46
s13207.1	7:43	14:43	23:43	32:43	43:43	109:42
s15850.1	6:25	16:25	27:25	39:25	53:25	132:25

Each entry → Time taken to extract the states : Max depth reached

6.5 Summary

In this chapter, we have presented a new SAT-based framework for state-space traversal that combines depth-first bounded reachability analysis together with bounded model checking. Experimental results show that our technique complements conventional bounded model checking in tracking down targets and can reach states beyond the bounds that can be reached by practical solvers. This enabled more than an order of magnitude runtime improvements than existing approaches.

There are several directions for future work. First, due to its search in a depth-first manner, our technique does not guarantee the minimal-length counter-example. More sophisticated heuristics might be employed to direct the search, for example, by using the functionality of the design. Second, there is no obvious evidence for setting an appropriate *depth* for performing the bounded reachability analysis. The depth of interest would be the one that can reject many states thereby improving the overall runtime. Finally, it would be interesting to incorporate techniques such as interpolation [70] and proof-based abstraction [69] within our bounded model checker and see the effectiveness as more initial states are added.

Chapter 7

Static Invariant Extraction

In this chapter, we propose a novel framework that statically extracts invariants of a synchronous sequential circuit and uses them to enhance SAT-based induction runs. First, we quickly generate a number of candidate invariants. Among these, many of the false and redundant candidates are eliminated via two low-cost techniques. For the remaining candidates, we employ a SAT-based induction method to establish an inductive-proof. All the candidates that are proved to be invariants are converted into constraint clauses which are applied in all the transition relations during the SAT checks for a given safety property.

7.1 Motivation

Before going to the motivation part, first consider the result of using induction to verify a property ϕ as given in Table 7.1. There exists three scenarios: (i) ϕ is of Type 1 if the base case is satisfiable, (ii) ϕ is of Type 3 if both the base case and the induction step runs are unsatisfiable and (iii) ϕ is of Type 2 if the base case is unsatisfiable but the induction step is satisfiable. Note that the notion of Type2 is with respect to a given depthi k. As the depth k is increased, ϕ might be classified into Type1 or Type3.

In the induction-step run, note that the initial-state is not constrained to any of the reachable state(s). When verifying a true safety-property, we want this induction-step run to be unsatisfiable at as smaller depth as

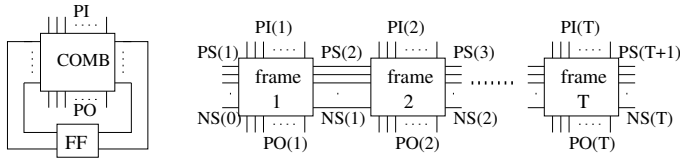
Table 7.1. SAT Runs for ϕ at depth k

Type	Base Case	Ind Step	Result
1	SAT	-	ϕ is not an invariant
2	UNSAT	SAT	Nothing concluded. Increase k
3	UNSAT	UNSAT	ϕ is an invariant

possible. However, since there are no initial state constraints, these runs tend to be satisfiable most of the time. This has been observed in the case of several industrial circuits as well [8].

If the formula for the induction step at a depth k is satisfiable, then the satisfying solution forms a simple path s_1, s_2, ..., s_{k+1} such that the first k states satisfy ϕ but the last state s_{k+1} does not. In such a scenario, two cases are possible: (i) the first state s_1 is a reachable state, and hence ϕ is definitely not an invariant. (ii) the first state s_1 is an illegal state in which case the counter-example obtained is *spurious* and nothing can be concluded about ϕ. However, in general, it is hard to identify the reachability of s_1. In [80], the authors assume s_1 to be an illegal state and obtain a strengthened property $\phi \wedge \bar{s_1}$. The main intuition behind it is that the new property is stronger than the original one and might be proved at an earlier induction-depth. However, if the induction run has several such spurious counter-examples (which can be exponential in the number of state-elements), the new property obtained might still require a very high induction depth to be proved.

From the above discussion, the induction-step runs can be improved by learning as many invariants as possible and restricting the SAT search with these invariants. For example, if we learn sufficiently many invariants that can void *all* the spurious counter-examples for a given safety property, then we would be directly proving that property. Another important benefit behind learning of such invariants is that they can be repeatedly asserted in all the unrolled transition relations, which further constrains the search close to the reachable space. Since a thorough reachability analysis (even over-approximate analysis using unbounded model checking) can be prohibitively expensive, we try to extract invariants of a sequential design via an alternative analysis. Whereas our technique does not guarantee the learning of *all* the possible invariants



(a) Sequential Circuit (b) Circuit Unrollment for T time−frames

Figure 7.1. Sequential circuit and its expansion

in a circuit, it identifies many temporal ones *i.e.* those which relates circuit signals across different clock-cycles. Such invariants might be useful for a better understanding of the circuit behavior and for proving other temporal properties.

7.2 Static Invariant Extraction

In this section, we provide our contributions and algorithms for candidate generation, pruning and verification. In all the following descriptions, we consider a transition relation in the form of a synthesized sequential circuit consisting of Boolean gates and flip-flops. A typical sequential circuit with primary inputs (PIs), primary outputs (POs) and flip-flops (FFs) is shown in Figure 7.1(a). A circuit unrolled for T frames is shown in Figure 7.1(b), where the next-state(NS) elements of previous frame are fed to the present-state(PS) elements of the next frame.

As mentioned before, we wish to extract as many invariants as possible via static analysis to improve SAT-based induction. In this work, we try to automate this at a gate-level description of a sequential circuit. We represent a logic value assignment, $v \in \{0, 1\}$, to a gate G in time-frame t with $G_{v.t}$. For each gate in a time-frame, there exists two assignments, one with logic 0 and the other with logic 1. Each candidate, ϕ, is represented as a disjunction of such gate-value assignments in the circuit. The complement, $\bar{\phi}$ is simply the conjunction of the opposite gate-value assignments. For example, if $\phi = G1_{v1.t1} \vee G2_{v2.t2} \vee ... \vee Gn_{vn.tn}$, then $\bar{\phi} = G1_{\overline{v1}.t1} \wedge G2_{\overline{v2}.t2} \wedge ... \wedge Gn_{\overline{vn}.tn}$, where $v1, v2, .., vn \in \{0, 1\}$ and $t1, t2, .., tn \in \{1, 2, .., T\}$. We refer to the number of elements in each candidate as the *size* of that candidate. The first stage of our framework is the candidate generation process which is given below.

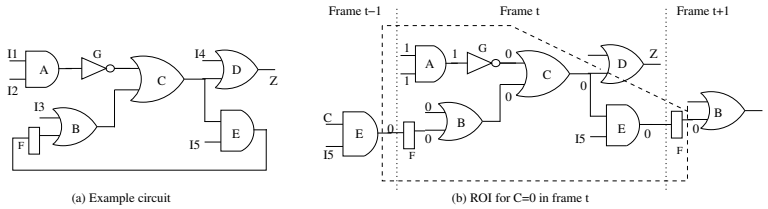


Figure 7.2. Region Of Influence for a Gate-value assignment

7.2.1 Generation of Candidate Invariants

In this section, we use two models, namely the Region-based model and Cut-based model, for generating the candidate invariants. In the Region-based model, we employ logic implications[1] to identify the suitable candidates. For each gate-value assignment, $G_{v.t}$, we define the Region Of Influence (ROI) as the topological circuit region within which the logic implications of $G_{v.t}$ encompass. Figure 7.2(a) shows an example circuit with one flip-flop F. Consider a gate-value assignment $C_{0.t}$ as shown in Figure 7.2(b). The logic implications of $C_{0.t}$ include $G_{0.t}$, $A_{1.t}$, $I1_{1.t}$, $I2_{1.t}$, $B_{0.t}$, $I3_{0.t}$, $F_{0.t}$, $E_{0.t-1}$, $E_{0.t}$ and $F_{0.t+1}$. All these assignments form the ROI for $C_{0.t}$. Now, consider the outer connected region for this ROI, which includes output of gate C and $I5$ in frame $t-1$, output of gate D in frame t and output of gate B in frame $t+1$. We call this signal set as the Outer Region of Influence (OROI) of $C_{0.t}$. Formally put, an OROI consists of the unspecified fanins of gates in ROI and the output signals of unspecified gates which has at least one fanin in ROI. The OROI for a conjunction of gate-value assignments can be obtained by logic implying all those assignments together in the circuit and then gathering the ROI.

Steps 1 through 11 of Algorithm 1 describes the candidate generation process via Region-based model. The algorithm takes as input the maximum number of time-frames (T) to be considered for propagating the logic implications. We form an initial set of candidates of size 1 using each gate in the center time-frame τ (step 4). Note that, here, we need to consider gate-value assignments only to the four primitive gates AND, NAND, OR and NOR, since assignments to other gates such as inverters and buffers are automatically covered by these. We employ these initial candidates as seeds and recursively generate larger-sized candidates using their OROIs. Given a candidate seed ϕ, we form $2 \times n$ new candidates ($\phi \vee G_{v.t}$) where each $G_{v.t} \in$

[1]Logic implications of $G_{v.t}$ are those gate-value assignments that are set when $G_{v.t}$ is set to true

$OROI(\bar{\phi})$, $v \in \{0,1\}$ and $|OROI(\bar{\phi})| = n$. For example, given a seed candidate $C_{1.t}$ in Figure 7.2(b), we compute OROI($C_{0.t}$) and obtain the corresponding 8 new candidates. The reason behind this kind of candidate selection is to explore the unspecified region beyond the region-of-influence. Note that if the seed candidate itself is an invariant, so will be its successors.

Next, in the Cut-based model, we use a cut in the circuit to obtain the candidates. The idea here is to gather potential candidates that might not be covered in the Region-based model. In the current framework, we use the cut comprised of all the flip-flops. Other cuts such as the min-cut(s) or a designer specified cut can also be used in addition to this. Once the cut signals are obtained, they are ordered so that each signal in the order is structurally near to its adjacent ones. We employ the MLP procedure [84] for this purpose. The MLP procedure first computes the input-support for each (next-state) flip-flop and then orders them such that the dependency matrix attains a block-triangular form. We refer the reader to [84] for its implementation details. Once the signals in the cut are ordered, we enumerate all possible combinations among groups of adjacent signals and form the candidates using their disjunctions. In contrast to the region-based model, all the elements in each candidate here belongs to the same time-frame. The pseudo-code for the Cut-based candidate generation is given in steps 12 through 18 of Algorithm 1, which are self-explanatory.

7.2.2 Pruning the Candidate Set

Depending on the size of the circuit and the size-limits supplied, the number of candidates generated in Algorithm 1 can be many. In this section, we provide two low-cost techniques to quickly prune the initial candidate list so that the computational complexity in verifying them in the next (third) stage is minimized.

In the first technique, we use the logical dependencies among the candidates to prune the redundant ones. If $\phi 1$ and $\phi 2$ are two candidates and if $\phi 1 \rightarrow \phi 2$, then $\phi 2$ is covered by $\phi 1$ and hence can be removed. This is because, if $\phi 1$ is an actual invariant, so will be $\phi 2$ and hence we need not check $\phi 2$. However, note that the reverse is not true, *i.e.*, it is possible for $\phi 2$ to be an invariant, but not for $\phi 1$. Nevertheless, each such $\phi 1$ can cover several other candidates, all of which can also be pruned out. This gives rise to a trade-off between the verification runtime required and the amount of learning that can be obtained. In our current framework,

Algorithm 6 Candidate Generation

1: Given odd T, $\tau = (T-1)/2$; Initialize CAND-LIST = $NULL$
2: Given Max Region-based candidate size, N; set $n = 1$
3: **while** ($n \leq N$) **do**
4: if $n = 1$, CAND-LIST = All $G_{0.\tau}, G_{1.\tau}$: GateType(G) $\in$ {AND, NAND, OR, NOR}
5: else // steps 6 through 9
6: **for** each candidate $\phi \in$ CAND-LIST with size $n-1$ **do**
7: LogicImply($\bar{\phi}$); Obtain $OROI(\bar{\phi})$
8: Form new candidates, ($\phi \vee G_{v.t}$) $\forall\, G \in OROI(\bar{\phi})$, $v \in \{0, 1\}$
9: **end for**
10: $n = n + 1$
11: **end while**
12: Given Max Cut-based candidate size, M
13: Order the cut signals using MLP // currently flip-flop cut
14: **for** each signal S_1 in ascending order **do**
15: $S_2, S_3, ..., S_M$ = Next $M-1$ signals in order
16: Enumerate all 2^M combinations among $S_1, S_2, ..., S_M$
17: Add disjunction of each enumeration to CAND-LIST; Remove S_1 from the order
18: **end for**

we allowed the above pruning to lower the computational costs. In practice, given enough time, one can still check for the invariance of $\phi 2$ and eliminate it only if $\phi 1$ is proved invariant.

Since learning $\phi 1 \rightarrow \phi 2$ is not straightforward, we individually check each of the elements of $\phi 1$ and $\phi 2$ for cover. Let $\phi 1 = X1_{vx1.tx1} \vee X2_{vx2.tx2} \vee ... \vee Xn_{vxn.txn}$ and $\phi 2 = Y1_{vy1.ty1} \vee Y2_{vy2.ty2} \vee ... \vee Yn_{vyn.tyn}$, both having size n. If $X1_{vx1.tx1} \rightarrow Y1_{vy1.ty1}$, $X2_{vx2.tx2} \rightarrow Y2_{vy2.ty2}$, ..., $Xn_{vxn.txn} \rightarrow Yn_{vyn.tyn}$, then we can conclude $\phi 1 \rightarrow \phi 2$ and remove $\phi 2$. The pseudo-code for this is shown in Algorithm 2 (steps 1 through 8). Note that, due to the recursive nature of the candidate generation process, we can only eliminate those having equal sizes. Otherwise, we will simply end up with candidates with size 1, since all the candidates of size n are automatically covered by those having sizes $< n$.

Algorithm 7 Candidate Pruning

1: **for** each candidate $\phi \in$ CAND-LIST **do**
2: Let $\phi = X1_{vx1.tx1} \vee X2_{vx2.tx2} \vee ... \vee Xn_{vxn.txn}$
3: Si = Set of gate-value assignments due to LogicImply($Xi_{vxi.txi}$)
4: **for** each candidate $\phi^{'} \in$ CAND-LIST **do**
5: Let $\phi^{'} = Y1_{vy1.ty1} \vee Y2_{vy2.ty2} \vee ... \vee Ym_{vym.tym}$
6: If $m = n$ and if each $Yi_{vyi.tyi} \in$ each Si, remove $\phi^{'}$
7: **end for**
8: **end for**
9: Given vector limit, N; $n = 1$
10: **while** (At least one candidate removed) or ($n < N$) **do**
11: Generate Random Vector V_n; LogicSimulate(V_n)
12: Remove all ϕ such that $\bar{\phi}$ is achieved
13: $n = n + 1$
14: **end for**

After the above candidate pruning process, the remaining candidates can belong to any of Type1, Type2 or Type3 described in Table 7.1. Among these the ones that we are interested in are Type3 (invariants that can be proved at a given depth). Whereas no direct technique exists for pruning out Type2 candidates, Type1

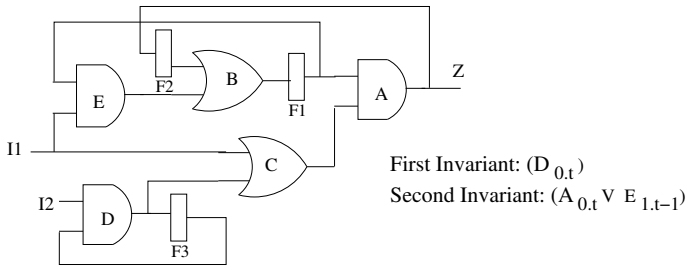


Figure 7.3. Example Invariant Extraction

candidates can be eliminated if the base case is satisfiable. In other words, if ϕ is a candidate and $\bar{\phi}$ can be reachable from the initial state(s), then ϕ is definitely not an invariant. In our framework, we employ random vector simulation to filter those Type1 candidates. Steps 10 through 14 of Algorithm 2 gives the pseudo-code. Essentially, we start with one known initial state. A random vector is applied at the primary inputs and logic simulated on the circuit. All the violated candidates are removed from the candidate list. The resulting state-assignment after each logic simulation is carried over as the initial state for the next random vector. This process is repeated for a given number of random vectors or until no more candidates are pruned in an iteration. The above process results in eliminating many of the false (Type1) candidates which need not be carried to the verification stage.

7.2.3 Verification of Candidates

After the end of above pruning process, we end up with a set of candidates which are highly probable to be Type2 or Type3. Given a maximum induction depth N (user-specified), we first conduct the base case run for each candidate. If this run is satisfiable for any candidate, then that candidate is false and hence eliminated. We use the corresponding satisfying solution to check if any other candidates are also violated. If so, all such candidates are immediately removed to save the computation time incurred in verifying them later. This process is given in steps 1 through 6 of Algorithm 3.

After the base-case run, all the Type1 candidates within N are removed and we are left with Type2 or Type3. Now, we can proceed directly to the induction step run with the maximum induction depth N for each of

these candidates. However, as mentioned earlier, the induction depths needed for certain candidates can be exponentially reduced when other known invariants are applied. Consider the example circuit shown in Figure 7.3 which has 3 flip-flops ($F1, F2, F3$) and 2 primary inputs ($PI1, PI2$). First, it can readily seen that $D_{0.t}$ is an invariant since the flip-flop $F3$ can never achieve a logic 1. The induction depth needed to prove this invariant is just 1. Now consider the single-gate candidate $A_{0.t}$. The ROI of $A_{1.t}$ includes $B_{1.t-1}$ and hence $A_{0.t} \vee E_{1.t-1}$ is generated as a candidate of size 2. This candidate is a true invariant and requires an induction depth of 8 to be proved. However, if we assert the invariant that $D = 0$ in each transition relation, the above candidate can be proved at a depth of just 1. Furthermore, this second invariant is a temporal one and asserting it together with the first one might prove more invariants, that are temporal in nature, in a larger circuit.

In other words, using the already proved invariants as constraints leads to more effective results. For this reason, rather than directly conducting the induction step runs at the maximum depth N, we start with the depth of 1 and then progress toward N. The pseudo-code for this process is shown in steps 7 through 15 of Algorithm 3. If the induction step for a candidate ϕ at step n is unsatisfiable, we conclude it as an invariant, remove it from the list and add it as a clause to the database C. At the same time, all other larger candidates that are covered by ϕ are also eliminated. For example, if $G_{v.t}$ is an invariant, so will be any other candidate containing elements implied by $G_{v.t}$. All the invariants learned so far are added to the CNF formulas when checking the other candidates for efficiency. The conversion of an invariant to a clause is straightforward since the candidates are already in disjunctive form.

7.3 Experimental Results

We have developed a prototype for candidate generation and pruning in C++ and integrated it with the zChaff SAT solver (version 2004.11.15) [85] for performing both the invariant extraction and the induction runs for the safety properties. Experiments for various sequential circuits from the ISCAS89 and ITC99 benchmark suites were conducted on a 3.2GHz Pentium 4 machine with 1GB RAM and running Linux OS.

First, we report the results for extracting the invariants in Table 7.2. To limit the initial number of candidates

Algorithm 8 Candidate Verification

1: Given Max Induction depth, N
2: **for** each candidate $\phi \in$ CAND-LIST **do**
3: $\bar{\phi}_{any} = \bar{\phi}_1 \vee \bar{\phi}_2 \vee ... \vee \bar{\phi}_N$
4: SAT-CHECK($I(1) \wedge T^*(1, 2, .., N) \wedge \bar{\phi}_{any}$)
5: If SAT, remove ϕ; Use solution to prune other false candidates
6: **end for**
7: Clause database $C = NULL$
8: **for** each n from 1 to N
9: **for** each candidate $\phi \in$ CAND-LIST **do**
10: $C^*(1, 2, .., n) = C_1 \wedge C_2 \wedge ... \wedge C_n$
11: SAT-CHECK($T^*(1, 2, .., n) \wedge \phi^*(1, 2, .., n-1) \wedge \bar{\phi}_n \wedge C^*(1, 2, .., n)$)
12: If UNSAT, remove ϕ and all other candidates covered by ϕ; Add ϕ as a clause to C
13: **end for**
14: **end for**
15: Clauses in C are the learned invariants

Table 7.2. Results for Invariant Extraction

Circuit	Gates	Initial Cand.			After Init. Pruning			Verified Cand.				Time
		Comb	Seq	Total	Comb	Seq	Total	Comb	Seq	Total	%Seq	
b10	189	8068	4799	12997	45	63	108	7	16	23	69.5	2
b13	362	7308	8178	15486	384	2117	2501	161	447	608	73.5	17
s526	368	12540	12528	25068	321	1035	1356	76	164	240	68.3	22
b11	396	18252	15097	33349	217	538	755	40	108	148	72.9	12
b07	433	19741	18534	38275	508	2993	3501	53	281	334	84.1	25
b04	546	24683	15823	40506	131	5290	5421	131	5289	5420	97.6	9
s832	622	65336	17082	82418	132	291	423	0	20	20	100	14
b05	642	21135	10156	31291	757	1512	2269	369	509	878	57.9	64
s1423	779	12659	10064	22723	454	1001	1455	43	55	98	56.1	31
b14	3708	882214	590372	1472586	1211	14375	15586	613	427	1040	41.1	426
s9234	6331	300994	94226	395220	1268	7560	8828	92	700	792	88.4	221
s13207	9517	436368	129511	565879	5249	31562	36811	309	1937	2246	86.2	449
s15850	11055	221002	99997	320999	4197	25420	29617	1066	4455	5521	80.7	402

Comb: Num combinational candidates Seq: Num sequential(temporal) candidates

obtained, we used a maximum region-based candidate size of 2 and a maximum cut-based candidate size of 3. The maximum induction depth is set to 5 for smaller circuits (< 1000 gates) and 3 for the larger ones. For each circuit, the column *Gates* shows the number of gates in it. The remaining columns report the number of initial candidates, those remaining after initial pruning and those verified as invariants. The columns *Comb* and *Seq* report the number of corresponding combinational and sequential (*i.e.* temporal) candidates. The column *%Seq* gives the percentage of sequential candidates among the total verified candidates. Finally the column *Time* reports the run time in seconds.

It can be observed that the number of initial candidates tend to be very large. This number grows with the circuit size and with the size-limit of each candidate. However, in the pruning stage, many of the false and redundant ones are quickly eliminated, thus leaving only a set of candidates with higher probability. One interesting thing to notice in most of these circuits is that, the amount of pruning in combinational candidates is much higher than that of the sequential ones. This might be due to the fact that we use simple

Table 7.3. Results for Induction Runs for Safety Properties

Circuit.ϕ	Gates	Original	Simp2C [74]		EBL [75]		Incr [7]	OURS		
		T_t	T_{ie}	T_t	T_{ie}	T_t	T_t	T_{ie}	T_t	Proved?
b10.1	189	83	0	118	1	8	5	2	**3**	Yes[25]
b10.2	189	86	0	131	1	7	14	2	**3**	Yes[25]
b13.1	362	534	0	89	1	*13*	86	17	18	No
s526.1	368	3565	0	3434	1	1291	841	21	**22**	Yes[25]
s526.2	368	4034	0	4386	1	1599	1451	21	**22**	Yes[50]
b11.1	396	5154	1	4864	1	3405	1471	12	**13**	Yes[50]
b11.2	396	5575	1	4800	1	2386	1053	12	**13**	Yes[25]
b07.1	433	193	0	42	1	*11*	52	25	71	No
b07.2	433	219	0	43	1	20	42	25	**26**	Yes[25]
b04.1	546	580	0	563	0	491	133	9	**10**	Yes[25]
b04.2	546	768	0	491	0	465	128	9	**10**	Yes[25]
s832.1	622	610	0	542	1	339	*145*	14	234	No
s832.2	622	272	0	190	1	31	33	14	*29*	No
b05.1	642	151	0	46	0	41	60	64	**65**	Yes[25]
b05.2	642	192	0	32	0	44	45	64	**66**	Yes[50]
s1423.1	779	7371	0	7616	1	8942	1651	31	**32**	Yes[25]
s1423.2	779	11107	0	8324	1	3798	1379	31	**32**	Yes[25]
b14.1	3708	597	3	505	48	101	270	426	**427**	Yes[10]
b14.2	3708	1357	3	1110	48	594	474	426	**427**	Yes[10]
s9234.1	6331	28015	1	21912	18	3033	2836	221	**223**	Yes[20]
s9234.2	6331	21319	1	18682	18	6252	2364	221	**222**	Yes[10]
s13207.1	9517	-TO-	2	28613	108	10712	5558	449	**450**	Yes[10]
s13207.2	9517	-TO-	2	-TO-	108	8258	11451	449	**451**	Yes[20]
s15850.1	11055	-TO-	3	-TO-	35	12092	*3689*	402	13806	No
s15850.2	11055	-TO-	3	-TO-	35	16019	*3778*	402	18570	No

(i) None of Original, Simp2C, EBL, and Incr could complete proof for any property

(ii) Smallest run-times for those proved properties are **bold-faced**

(iii) Smallest run-times for those unproved properties are *italicized*

(iv) T_{ie}: Invariant extraction time T_t: Total time (with T_{ie}) *TO*: $T_t > 30000$

logic implications for pruning in Algorithm 2, which simply might not extend into the other time-frames. For the same reason, the number of initial combinational candidates tend to be higher than that of the sequential ones. Next, when we look at the number of verified candidates, it averaged around 21.4% of the candidates remaining after pruning. In other words, 78.6% of them belong to either Type1 or Type2. Among these, we observed that 70 to 90% are Type2, even though the exact numbers are not reported in the table due to space limitations. This is encouraging, because, given more time resources, one might extract more invariants out of these Type2 candidates at greater unrolled depths.

Consider the circuit b04. In this circuit, 5420 out of the 5421 candidates were proved to be invariant. In other words, there was almost no execution time wasted in verifying the Type1/Type2 candidates. Finally, when we looked at the percentage of sequential candidates among the total verified ones, the average turned out to be 76.5%. This is significant because such invariants enable a better understanding of the design and may be helpful in proving other temporal safety properties. Within the total run-time, less than 5% was spent by Algorithms 1 and 2 during candidate generation and pruning, whereas the rest was spent in Algorithm 3 for candidate verification.

Next, we report the results for safety-property verification by embedding the extracted invariants. The experiments were conducted for hard safety properties, which were based on aborted partial-states[2] from a sequential ATPG. An aborted state is a state that the ATPG failed to justify within its resource limit. If s is an aborted partial-state, the property was formulated as $AG(\bar{s})$. For smaller circuits, we increased the induction depth k by 25 frames per iteration up to a maximum of 250 frames. Whereas for the larger ones, these were set to 10 and 100 respectively. The time-out limit for each property was set to 30,000 seconds.

Table 7.3 reports the experimental results. We compare our technique with the original induction run (*i.e.*, without any learning), learning from Simp2C [74] and EBL [75] and Incremental learning as done in [7]. All the invariants learned are replicated as clauses in all the transition relations. For each property instance shown, we report the induction run-times required by the corresponding techniques, where T_{ie} and T_t show the invariant extraction and the total (T_{ie} + solving) run times, respectively, in seconds.

[2] A partial-state is a set of assignments to a group of flip-flops (not necessarily all of them)

As a first observation, we can see that the invariant extraction times needed by our method is in general higher than that of others (Simp2C and EBL). However, in most cases, they are small when compared with the actual induction run-times. If a property is proved as an invariant via our learning, we report the corresponding bound in brackets under the column *Proved*. Among the 28 properties tested, we were able to complete the proof for 22 properties. In contrast, all the other approaches could not prove any of the 28 properties. For example, for s13207.1, the original run, Simp2C, EBL and Incremental learning took more than 30000, 28000, 10000 and 5000 seconds, respectively, without proving the property; on the other hand, we verified it as an invariant in only 450 seconds. Note that in proving these properties, we have achieved more than an order of magnitude improvement in run-times. For the cases where none of the techniques could prove the property, our overall runtime tended to be higher than techniques. This might be due to two reasons: If these properties are not actually invariants, our added clauses might not be as efficient as the other techniques in pruning the combinational search-spaces. On the other hand, if these properties are indeed invariants, the extra clauses we add prunes many of the satisfying solutions and hence finding one solution among the remaining less number of solutions may need more time. Nevertheless, in all these unproved cases, our total run-time was smaller than that of the *original* run-time, displaying the effectiveness of the added clauses.

7.4 Summary

We have proposed a novel framework that automates the extraction of a set of invariants of a sequential design and applies them to enhance SAT-based induction. All the learned invariants are converted to constraint clauses and inserted into all transition relations to restrict the SAT search as close as possible to the reachable space. Experiments revealed that our learning can prove many properties at early induction depths where as the conventional runs failed. This lead to more than an order of magnitude savings in the property checking run-times.

There are two possible directions for future research. In our current framework, all the invariants are computed before-hand irrespective of the property. In practice, one might dynamically compute only those required for a given property. For example, the cone-of-influence (COI) of a property might be used to reduce

the number of initial candidates. Secondly, it will be interesting to explore the candidates for k^{th}-invariants. For example, some of the Type1 candidates that we eliminate might actually turn out to be invariants after a given number of clock-cycles.

Chapter 8

Conclusions and Future Work

In this dissertation, we have outlined the main limitations in SAT-based formal verification approaches including SAT-based Bounded Model Checking and SAT-based Induction. These can be summarized as follows: (i) Lack of ability of the SAT solver to learn internal variable relationships, (ii) Exponential increase in run-time at higher depths that prohibits complete verification of a property, (iii) Lack of knowledge about non-trivial invariants of a design, and (iv) Lack of information about co-invariants for a given property.

In Chapter 3, we have proposed two new techniques for increasing the deductibility in CNF instances for enhancing SAT-based Bounded Model Checking. All the logic dependencies learned via those techniques are converted to constraint clauses and replicated in the transition relations during the SAT runs. Experimental results on a variety of circuits and properties have shown that significant improvements in runtime can be consistently achieved. Whereas implication learning helps in speeding-up the SAT runs, it doesnt enable the reduction in depths required to prove a property. There are several directions for future work. First, note that in some cases, the solving times actually increase due to the added clauses. This is because, due to the added clauses, the number of occurrences of each variable is increased, which leads to a possibly different decision order chosen by the SAT solver. It will be interesting to study the effect of added clauses upon the variable order and research a better ordering strategy. Secondly, all the clauses being added might not be useful in the same way. For example, some clauses might be trivial and can be easily learned by the

SAT solver, given the other clauses. Careful elimination of such clauses might reduce the burden on the SAT solver and avoids overheads in BCP. Finally, new ways to learn non-trivial implications would definitely be a promising future-work.

Chapter 4 proposes novel technqiues for fast identification of illegal state spaces. We employed ATPG based learning, exploited logic implications and used an enumerative framework to learn more illegal states. All the learned states are used as constraints at the state-boundaries in order to improve the SAT-based induction runs. Experiments have shown that these constraints often reduce the run-times and at the same time can reduce the depth required to prove a property thereby achieving more than an order of magnitude runtime improvements. One limitation of this technique is that it can only be helpful for true properties and does not have the ability to reduce the depth for false properties. There are two directions for future work. First, the above framework aims at *low-cost* and might miss crucial illegal states. For further hard to solve properties, it would be interesting to generalize this learning to capture additional illegal states. Second, the learning regions can be limited according to a given property to reduce the pre-processing times further.

In Chapter 5, we proposed techniques using ATPG-based pre-image computations and co-invariant extractions to strengthen a property. Resolution and Dynamic variable ordering are further employed to improve the quality of strengthening. Once a property is strengthened, it can be verified using BMC/Induction similar to the original property. Experimental results have shown that the proof-depths can be reduced significantly thereby achieving several orders of magnitude runtime improvements. As opposed to the above two techniques, strengthening enables in proving both passing as well as failing properties faster. There are two important directions of future work. First, it can be seen that property strengthening can sometimes be a overhead due to the additional clauses added. Clever minimization of the added clauses would be interesting to pursue. For example, some of the added clauses might not present any value for strengthening when certain other clauses are added. Secondly, it will be interesting to exploit ways of automatically identifying co-invariants for a given safety property as opposed to pre-selecting them as done in our work.

In Chapter 6, we presented a novel approach that augments bounded reachability analysis with bounded model checking. The main idea here is to learn a sequence of states, from the initial state(s) in the forward direction and from the target state(s) in the backward direction, and use bounded model checking to check

if atleast one backward state can be reached from atleast one forward state. Such a framework is very powerful because of the exponential number of new states it can cover compared to conventional bounded model checking. While doing so, for true properties, the strengthened target states can also be proved earlier via induction runs. Comparison with state-of-the-art incremental learning techniques and sequential SAT solvers showed that our technique out-performs them in most cases by providing the result faster. There are few directions for future work. First, due to its search in a depth-first manner, our technique does not guarantee the minimal-length counter-example. More sophisticated heuristics might be employed to direct the search, for example, by using the functionality of the design. Second, there is no obvious evidence for setting an appropriate *depth* for performing the bounded reachability analysis. The depth of interest would be the one that can reject many states thereby improving the overall runtime. Finally, it would be interesting to incorporate techniques such as interpolation [70] within our bounded model checker and see the effectiveness as more initial states are added.

Finally, in Chapter 7, we proposed two models for extracting candidate invariants and using induction to prove them. The region-based model aims at using logic implications to obtain a set of non-trivial candidates in the circuit, where as the cut-based model aims at obtain candidates at a given cut in the circuit (say, the flip-flop cut). Once the candidates are obtained, the false ones among them are eliminated and the proof for the remaining ones is established using induction. All the invariants that were proved can be used as constraints during the induction runs which will help in proving complex or difficult properties. There are two possible directions for future research. In our current framework, all the invariants are computed before-hand irrespective of the property. In practice, one might dynamically compute only those required for a given property. For example, the cone-of-influence (COI) of a property might be used to reduce the number of initial candidates. Second, it will be interesting to explore the candidates for k^{th}-invariants. For example, some of the Type1 candidates that we eliminate might actually turn out to be invariants after a given number of clock-cycles.

Bibliography

[1] N. H. E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, Third Edition, 2005.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 317–320, 1999.

[3] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties using Induction and a SAT Solver," in *Proceedings of International Conference on Formal Methods in Computer Aided Design*, pp. 108–125, 2000.

[4] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment," in *Proceedings of Conference on Correct Hardware Design and Verification Methods*, pp. 254–268, 2005.

[5] N. Amla, R. Kurshan, K. McMillan, and R. Medel, "Experimental Analysis of Different Techniques for Bounded Model Checking," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 34–48, 2003.

[6] O. Strichman, "Accelerating bounded model checking of safety properties," *Formal Methods in System Design*, vol. 24, pp. 5–24, Jan 2004.

[7] N. Een and N. Sorensson, "Temporal Induction by Incremental SAT Solving," in *International Workshop on Bounded Model Checking*, July 2003.

[8] L. Zhang, M. R. Prasad, and M. S. Hsiao, "Incremental Deductive & Inductive Reasoning for SAT-based Bounded Model Checking," in *Proceedings of International Conference on Computer Aided Design*, pp. 502–509, 2004.

[9] O. Shacham and E. Zarpas, "Tuning the vsids decision heuristic for bounded model checking," in *International Workshop on Microprocessor Test and Verfication*, 2003.

[10] B. Li and F. Somenzi, "Efficient Computation of Small Abstraction Refinements," in *Proceedings of International Conference on Computer Aided Design*, pp. 518–525, 2004.

[11] M. N. Velev, "Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 266–271, 2004.

[12] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Property Checking," in *Proceedings of International Conference on Computer Aided Design*, pp. 50–57, 2004.

[13] V. C. Vimjam and M. S. Hsiao, "Increasing the Deductibility in CNF Instances for Efficient SAT-based Bounded Model Checking," in *Proceedings of High Level Design, Validation and Test Workshop*, pp. 52–59, Nov 2005.

[14] V. C. Vimjam and M. S. Hsiao, "Fast Illegal State Identification for Improving SAT-based Induction," in *Proceedings of Design Automation Conference*, pp. 241–246, July 2006.

[15] V. C. Vimjam and M. S. Hsiao, "Static Invariant Extraction for Fast Verification of True Properties," in *To be submitted*, 2007.

[16] V. C. Vimjam and M. S. Hsiao, "Explicit Safety Property Strengthening in SAT-based Induction," in *Proceedings of International Conference on VLSI Design*, Jan 2007.

[17] V. C. Vimjam and M. S. Hsiao, "Augmenting Bounded Model Checking and Bounded Reachability Analysis," in *Submitted to an anonymous conference*, 2007.

[18] S. Cook, "The Complexity of Theorem Proving Procedures," in *ACM SIGACT Symposium on the Theory of Computing*, 1971.

[19] M. Davis and H. Putnam, "A computing procedure for quantification theory," *ACM Journal*, vol. 7, pp. 201–215, 1960.

[20] J. P. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *IEEE Trans. on Computers*, vol. 48, pp. 506–521, May 1999.

[21] H. Zhang, "SATO: An Efficient Propositional Prover," in *Proceedings of International Conference on Automated Deduction (CADE)*, 1997.

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT-Solver," in *Proceedings of Design Automation Conference*, pp. 530–535, 2001.

[23] E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust SAT-Solver," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 142–149, 2002.

[24] L. Ryan, "http://www.cs.sfu.ca/ loryan/personal," in *Siege SAT solver version 4*.

[25] N. Een and N. Sorensson, "Minisat a sat solver with conflict-clause minimization," in *International Conference on Theory and Applications of Satisfiability Testing, Poster*, 2005.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Proceedings of International Conference on Computer Aided Design*, Nov 2001.

[27] "htpp://www.satisfiability.org : International Conference on Theory and Applications of Satisfiability Testing,"

[28] "http://www.satcompetition.org : The International SAT Competition,"

[29] F. Bachhus and J. Winter, "Effective Preprocessing with Hyper-Resolution and Equality Reduction," in *Proceedings of Workshop on Satisfiability*, pp. 341–355, 2003.

[30] S. Subbarayan and D. Pradhan, "NiVER: Non-Increasing Variable Elimination Resolution for preprocessing SAT instances," in *Proceedings of Workshop on Satisfiability*, pp. X–Y, 2004.

[31] N. Een and N. Sorensson, "Effective preprocessing in sat through variable and clause elimination," in *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[32] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of International Conference on Computer-Aided Design*, pp. 534–537, 1993.

[33] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for Combinational Equivalence Checking," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 114–121, 2001.

[34] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to cad problems - test, verification and optimization," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1149–1158, Sept 1994.

[35] J. Marques-Silva and T. Glass, "Combinational Equivalence Checking using Satisfiability and Recursive Learning," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 145–149, 1999.

[36] Y. Novikov, "Local Search for Boolean Relations on the Basis of Unit Propagation," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. X–Y, 2003.

[37] F. Lu, L.-C. Wang, K.-T. Cheng, and R. C.-Y. Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 892–897, 2003.

[38] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna, "A Signal Correlation Guided ATPG Solver And Its Applications For Solving Difficult Industrial Cases," in *Proceedings of Design Automation Conference*, June.

[39] I.-H. Moon and C. Pixley, "Non-Miter-Based Combinational Equivalence Checking by Comparing BDDs with Different Variable Orders," in *Proceedings of Formal Methods in Computer-Aided Design*, 2004.

[40] M. Syal and M. Hsiao, "VERISEC: VERIfying Equivalence of SEquential Circuits using SAT," in *Proceedings of High-Level Design Validation and Test Workshop*, pp. 52–59, 2005.

[41] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton, "Theory of Safe Replacements for Sequential Circuits," in *IEEE Transcations on CAD of Integrated Circuits and Systems*, vol. 20, pp. 249–265, Feb 2001.

[42] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," in *IEEE Transactions on Computer Aided Design*, vol. 11, pp. 1469–1494, Dec 1992.

[43] K. T. Cheng, "Redundancy Removal for Sequential Circuits Without Reset States," in *IEEE Transactions on Computer Aided Design*, vol. 12, pp. 13–24, Jan 1993.

[44] S. Y. Huang, K. T. Cheng, and K. C. Chen, "Verifying sequential equivalence using atpg techniques," in *ACM TODAES*, vol. 6, pp. 244–275, April 2001.

[45] F. Lu and K.-T. Cheng, "Sequential Equivalence Checking Based on K-th Invariants and Circuit SAT Solving," in *Proceedings of High Level Design Validation and Test Workshop*, pp. 45–51, 2005.

[46] F. Lu and K.-T. Cheng, "IChecker: An Efficient Checker for Inductive Invariants," in *Proceedings of High Level Design Validation and Test Workshop*, 2006.

[47] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic," in *ACM Trans. on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[48] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[49] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[50] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," in *IEEE Trans. on Computers*, vol. 35, pp. 677–691, 1986.

[51] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD package," in *Proceedings of Design Automation Conference*, pp. 40–45, 1990.

[52] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proceedings of International Conference on Computer Aided Design*, 1988.

[53] H. Fujii, G. Ootomo, and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," in *Proceedings of International Conference on Computer Aided Design*, pp. 38–41, 1993.

[54] P. Y. Chung, I. N. Hajj, and J. H. Patel, "Efficient Variable Ordering Heuristics for Shared ROBDD," in *Proceedings of ISCAS*, 1993.

[55] S. Panda and F. Somenzi, "Who are the variables in your neighbourhood," in *Proceedings of International Conference on Computer Aided Design*, 1995.

[56] M. Thornton, J. Williams, R. Drechsler, and N. Drechsler, "Variable re-ordering for shared Binary Decision Diagrams using output probabilities," in *Proceedings of Conference on Design Automation and Test in Europe*, 1999.

[57] F. Aloul, I. Markov, and K. Sakallah, "MINCE: A Static Global Variable-ordering for SAT and BDD," in *Proceedings of International Workwhop on Logic and Synthesis*, 2001.

[58] F. Aloul, I. Markov, and K. Sakallah, "FORCE: A Fast and Easy-To-Implement Variable Ordering Heuristic," in *Proceedings of Great Lakes Symposium on VLSI*, 2003.

[59] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on SAT-Solvers," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[60] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, "SAT-Based Image Computation with Application in Reachability Analysis," in *Proceedings of International Conference on Formal Methods in Computer Aided Design*, 2000.

[61] B. Li, M. S. Hsiao, and S. Sheng, "A Novel SAT All-Solutions Solver for Efficient Preimage Computation," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 272–277, 2004.

[62] K. L. McMillan, "Applying SAT methods in unbounded Symbolic Model Checking," in *Proceedings of International Conference on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science*, pp. 250–264, 2002.

[63] H. J. Kang and I. C. Park, "SAT-based Unbounded Model Checking," in *Proceedings of Design Automation Conference*, 2003.

[64] J. Baumgartner and A. Kuehlmann, "Enhanced Diameter Bounding via Structural Transformation," in *Proceedings of Conference on Design, Automation and Test in Europe*, 2004.

[65] M. K. Ganai, A. Gupta, and P. Ashar, "Beyond Safety: Customized SAT-based Model Checking," in *Proceedings of Design Automation Conference*, pp. 738–743, 2005.

[66] C. Wang, H. Jin, G. D. Hatchel, and F. Somenzi, "Refining the SAT Decision Ordering for Bounded Model Checking," in *Proceedings of Design Automation Conference*, pp. 535–538, 2004.

[67] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Learning from BDDs in SAT-based Bounded Model Checking," in *Proceedings of Design Automation Conference*, pp. 824–829, 2003.

[68] G. Cabodi, S. Nocco, and S. Quer, "Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 898–903, 2003.

[69] K. L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[70] K. L. McMillan, "Interpolation and SAT-based Model Checking," in *Proceedings of Conference on Computer Aided Verification*, pp. 1–13, 2003.

[71] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle, "Dynamic Abstraction Using SAT-based BMC," in *Proceedings of Design Automation Conference*, pp. 754–757, 2005.

[72] C. A. J. van Eijk, "Sequential Equivalence Checking without State-Space Traversal," in *Proceedings of Conference on Design, Automation and Test in Europe*, 1998.

[73] P. Bjesse and K. Claessen, "SAT-based Verification without State-Space Traversal," in *Proceedings of Conference on Formal Methods in Computer Aided Design*, 2000.

[74] R. Arora and M. S. Hsiao, "Enhancing SAT-based Bounded Model Checking using Sequential Logic Implications," in *Proceedings of International Conference on VLSI Design*, pp. 784–787, 2004.

[75] J. Zhao, M. Rudnick, and J. Patel, "Static Logic Implication with application to fast redundancy identification," in *Proceedings of VLSI Test Symposium*, pp. 288–293, 1997.

[76] A. Church, "Introduction to Mathematical Logic," in *Princeton, NJ: Princeton Univ.*, 1956.

[77] D. LeBerre, "Exploiting the Real Power of Unit-Propagation Lookahead," in *Proceedings of Workshop on Satisfiability*, 2001.

[78] C. M. Li, "Integrating Equivalency Reasoning into Davis-Putnam procedure," in *Proceedings of AAAI Conference*, pp. 291–296, 2000.

[79] J. Zhao, J. A. Newquist, and J. H. Patel, "A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-Cycle Redundancy Identification," in *Proceedings of International Conference on VLSI Design*, pp. 163–169, 2001.

[80] L. de Moura, H. Rueß, and M. Sorea, "Bounded Model Checking and Induction: From Refutation to Verification," in *Proceedings of Conference on Computer Aided Verification*, 2003.

[81] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, vol. C-30, pp. 215–222, Mar 1981.

[82] S. Sheng and M. S. Hsiao, "Efficient Preimage Computation Using a Novel Success-Driven ATPG," in *Proceedings of Conference on Design Automation and Test in Europe*, pp. 822–827, 2003.

[83] H. Goldstein and E. L. Thigpen, "Scoap: Sandia Controllability/Observability analysis program," in *Proceedings of Design Automation Conference*, 1980.

[84] I.-H. Moon, G. Hachtel, and F. Somenzi, "Border-block Triangular Form and Conjunction Schedule in Image Computation," in *Proceedings of Conference on Formal Methods in Computer Aided Design*, 2000.

[85] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 279–285, 2001.

[86] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property Checking via Structural Analysis," in *Proceedings of International Conference on Computer Aided Verification*, 2002.

[87] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, "Combining Strengths of Circuit-based and CNF-based algorithms for a High-Performance SAT Solver," in *Proceedings of Design Automation Conference*, pp. 747–750, June 2002.

[88] F. Lu, M. K. Iyer, G. Parthasarathy, and K.-T. Cheng, "An Efficient Sequential SAT Solver With Improved Search Strategies," in *Proceedings of Design Automation and Test In Europe Conference*, 2005.

Vita

Vishnu C. Vimjam was born on May 24, 1981 in Kavali, a small town in coastal Andhra Pradesh in India. He had his primary education in the Railway Mixed English Medium High School (Bitragunta - AP) till 1991 and then higher education in Ongole Public School (Ongole - AP) till 1996. He recieved the best out-going student award from Ongole Public School in 1996. He then joined Narayana Residential Junior College (Brahmadevam - AP) and completed his Intermediate college studies for two years. In 1998, he joined Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering and Technology (affiliated to Jawaharlal Nehru Technological University, Hyderabad - AP, India) and obtained his Bachelor of Technology Degree in Electronics and Communications Engineering in April 2002. In 2002, he recieved the Justice Punnaiah Gold Medal for his excellent academic achievements and also the best out-going male student award for his overall undergrad performance. Later, he joined Virginia Tech in Fall 2002 and was admitted into the direct PhD program in Computer Engineering in Spring 2004. He obtained his Masters Degree in Computer Engineering from Virginia Tech in Decemeber 2004 and is currently pursuing his PhD on SAT-based Formal Verification. His other research interests include BDD-based Model Checking and Sequential ATPG. After the completion of his PhD, he will be joining Real Intent, a company specializing in functional verification based in California.