Sequential Equivalence Checking of Circuits with Different State Encodings by Pruning Simulation-based Multi-Node Invariants

Zeying Yuan

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

Master of Science
in
Computer Engineering

May 5, 2015
Blacksburg, Virginia

Keywords: Sequential Equivalence Checking(SEC), Multi-node Inductive Invariants,
Constrained Logic Synthesis

Sequential Equivalence Checking of Circuits with Different State Encodings by Pruning Simulation-based Multi-Node Invariants

Zeying Yuan

(ABSTRACT)

Verification is an important step for Integrated Circuit (IC) design. In fact, literature has reported that up to 70% of the design effort is spent on checking if the design is functionally correct. One of the core verification tasks is Equivalence Checking (EC), which attempts to check if two structurally different designs are functionally equivalent for all reachable states. Powerful equivalence checking can also provide opportunities for more aggressive logic optimizations, meeting different goals such as smaller area, better performance, etc. The success of Combinational Equivalence Checking (CEC) has laid a foundation to industry-level combinational logic synthesis and optimization. However, Sequential Equivalence Checking (SEC) still faces much challenge, especially for those complex circuits that have different state encodings and few internal signal equivalences.

In this thesis, we propose a novel simulation-based multi-node inductive invariant generation and pruning technique to check the equivalence of sequential circuits that have different state encodings and very few equivalent signals between them. By first grouping flip-flops into smaller subsets to make it scalable for large designs, we then propose a constrained logic synthesis technique to prune potential multi-node invariants without inadvertently losing important constraints. Our pruning technique guarantees the same conclusion for different instances (proving SEC or not) compared to previous approaches in which merging of such potential invariants might lose important relations if the merged relation does not turn out to be a true invariant. Experimental results show that the smaller invariant set can be very effective for sequential equivalence checking of such hard SEC instances. Our approach is up to 20× faster compared to previous mining-based methods for larger circuits.

Acknowledgments

I want to express my sincere gratitude to my research adviser, Dr. Michael S. Hsiao, for his guidance and help during my research. I am really grateful for the opportunity working with him and I learnt so much from him both in my research and in my life. This work would have never been possible without his instruction for both the courses he provided or his intuitive suggestions during individual meetings. I would also want to thank Dr. Patrick Schaumont and Dr. Chao Wang for agreeing to be on my committee.

I will always remember the time that I had in PROACTIVE. I received a lot of help from all the lab mates that I really appreciate.

Finally, I want to thank my parents who love me, encourage me and have always been there for me. I also want to thank my boyfriend for his love, understanding and support. Meeting him is the best thing in my life.

Zeying

May 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The IC industry has progressed tremendously in the past few decades, delivering reliable, high performance, low power, and small area products which has dramatically changed people's life in many aspects: health care, finance, engineering, education, transportation and so on. According to Moore's Law, which says that the number of transistors on a single chip doubles roughly every 18 months, has resulted in extremely complex IC designs nowadays as shown in Fig. 1.1. In the plot, a bunch of released chips have been marked and the curve proves that transistor count doubling every two years. In 1971, Intel released its first commercially available microprocessor 4004 used for a calculator, which contained 2,300 transistors. Now, forty-some years later, its newest chip has more than 5 billion transistors, 2 million times larger compared with the 4004 microprocessor. Obviously, this drastic change of the size and complexity has not only shaped the design to a very strict and semi-automated industrial flow, but also caused a tremendous problem in verification, which attempts to ensure the correctness of design before the design is actually manufactured.

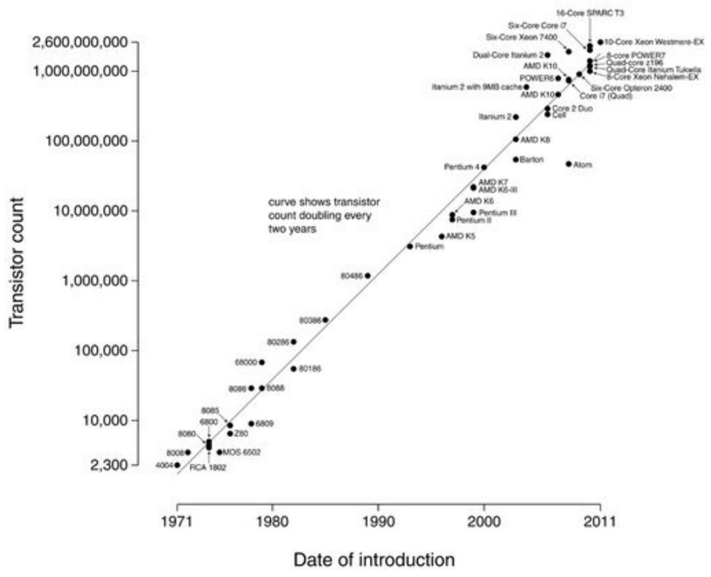


Fig. 1.1: Plot of Transistor Counts against Dates of Introduction [26], Plot of Transistor Counts against Dates of Introduction [26], Max Roser (2014) - 'Technological Progress'. Published online at OurWorldInData.org. Retrieved from: http://ourworldindata.org/data/technology-and-infrastructure/moores-law-other-laws-of-exponential-technologicalprogress/ Used under fair use, 2015.

Verification is becoming increasingly more difficult, especially for functional verification, in which 100% coverage often could not be guaranteed. In 2001, Andreas Bechtolsheim, Cisco Systems engineering vice president, was quoted in EE Times with one of the higher estimates: 'Design verification still consumes 80% of the overall chip development time'[1]. Verification of an integrated circuit design is not only an important component of the total effort but also a critical contributor to cost. Thus, any method that could lead to more efficient verification should be adopted aggressively.

One of the core verification tasks is equivalence checking, which attempts to check whether two structurally different designs are functionally equivalent or not for all reachable states. Powerful equivalence checking can also lead to more aggressive logic optimizations, meeting different goals such as smaller area, better performance, etc. SEC is a much more difficult problem due to the presence of state elements, or flip-flops, in which the circuit outputs depend not only on primary inputs but also on previous states. Consequently, we have to deal with a much larger search space. A lot of researches have been done in this field and they made much progress in recent years, but SEC still faces hurdles in dealing with hard SEC instances.

1.1 IC development flow

Fig. 1.2 shows a typical IC development flow. The flow begins with the system specification, which provides an overview of the functionality, in addition to non-functional aspects of the design such as die size, power consumption, speed as well as number of pins on the chip and so on. Following the specification, architectural design and functional design provide a behavior level description of the design. The design is then implemented using a Register Transfer Level (RTL) language which describe the structural information of the circuit. Many different strategies can be used to speedup the process, especially since the designs are generally complex and a lot of general purpose units have already been designed in the past and can be used as modules.

Between every two levels of abstraction, verification needs to be performed to make sure the design is bug-free and functional correctly. Designers might want to try different state encodings and optimization strategies with regard to area, power, or performance, thus it is also our interest to verify whether the two designs under checking are functionally equivalent or not. This type of verification is often called "equivalence checking". If the verification failed, we need to go back to upper level re-designing our system. When all levels have been verified, we could go to fabrication and perform other testing procedures. Obviously, verification is critical in the development overflow to produce reliable designs and may take the majority of total time and effort.

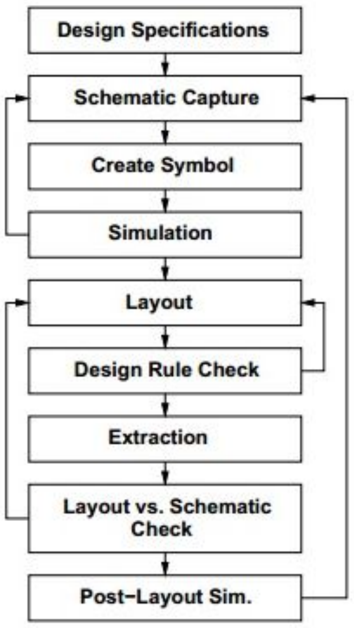


Fig.1.2 Major Steps in the IC design flow [27], http://techupdates.in/very-large-scale-integration-technology-and-its-design-flow/ Used under fair use, 2015.

1.2 Formal Verification

There exists two kinds of equivalence checking methodologies in IC industry: simulation-based verification and formal verification. Simulation-based verification requires one to enumerate input sequences and observe whether the circuits described at two different levels of abstraction lead the same outputs. While simulation could aid in some verification tasks, for proving equivalence of two designs, simulation is inadequate. Even exhaustive simulation is not a feasible solution as the number of states grows exponentially to the increasing number of flip-flops.

Formal verification is the act of proving or disproving the correctness of intended algorithm underlying a system with respect to a certain formal specification or property, using formal methods of mathematics [2]. It has been proved efficient in the systems such as cryptographic protocols, combinational circuits, sequential circuits and software expressed as source code. Although it has made much progress in recent years, formal verification still faces hurdles for large scale designs. The growth in design complexity increases the importance of formal verification techniques in hardware industry [3]. Sequential Equivalence Checking (SEC), as a significant field of sequential design, also benefits from formal verification. Basically, there are three basic tools and problem formulations in formal verification: Boolean Satisfiability (SAT), Binary Decision Diagram (BDD) and Automatic Test Pattern Generation (ATPG).

SAT is a well known approach to formal verification, which consists of an implicit exploration of the mathematical model of the circuit. In general, the circuit is converted into a propositional formula consisting of Boolean constraints. Then, it is passed onto a SAT solver. The process of solving is a search for a solution that satisfies all the given Boolean constraints. SAT is one of the first problems that were proven to be NP-complete by Stephen Cook and Richard Karp in 1970's, which means no algorithms is known that solve SAT efficiently and correctly [28][29]. However, many sequential circuit instances in practice, can actually be solved rather efficiently using heuristical SAT solvers. Since this thesis is based on SAT, we will discuss it in detail in the following chapters.

BDD is another significant approach to formal verification. Though the concepts of BDDs are relatively old [Lee59, Ake78], it was the work of Bryant [Bry86] that attracted the attention back and renewed the interest of many researchers [4]. Bryant observed that reduced, ordered, binary decision diagram (ROBDD) is a canonical representation of Boolean functions which means for a given total order of the variables, there is a one-to-one correspondence between functions and ROBDD. However, when constructing BDD to represent functions for system verification, the size of it can depend heavily on the ordering of input variables used to encode the function. Therefore, finding a good variable ordering is crucial [49]. The problem of finding an optimal variable ordering for constructing a minimum-size decision diagram is also known to be NP-complete [50]. So after Bryant's seminal work, many researches have been made in BDD field to make it more scalable, including approaches to find optimal variable ordering algorithm, BDD manipulation and constructing BDD more efficiency. Nowadays, verification of the correctness of hardware relies on BDDs for both the circuit representations and the manipulation of sets of states. For sequential equivalent checking (SEC), if two designs have isomorphic ROBDDs, we could conclude they are equivalent, which makes the test for equivalence inexpensive. Symbolic model checking algorithms based on BDDs have successfully verified properties of systems with very large numbers of states, verification and optimization of sequential circuits also benefit from the use of BDDs.

ATPG is also widely used in verification and testing field. It's an algorithm to automatically generate input patterns and explore all potential faults. It has already showed the power in different kinds of fault models, for example stuck-at-fault, which says that a signal in a design is stuck at a constant value due to some design or manufacturing defect, and bridge fault, in which 2 signals are shorted and may share the same logic value. Generally, given sufficient time, ATPG engine could find a solution to the problem or terminate with no-test-exist conclusion, which means it may achieve 100% completeness. Based on its property, ATPG could also be efficient in SEC. We could merge two target designs together and check whether the outputs are the same for all inputs patterns. Its complexity is similar to SAT Solver. Many different ATPG algorithms have been developed in recent decades to address combinational and sequential circuits like D Algorithm [44], the first complete test generation algorithm and Path-Oriented Decision Making (PODEM) [45], the first ATPG tool enumerating a binary decision tree. Efficient heuristics to prune the search space have further been proposed in [FS83, KM87, GB91, GF01]. Later work in ATPG mainly concentrated on effective implication procedures [SA89, RC90, KP92, CA93, TG00, GF01] [46].

1.3 Previous Work

In the past few decades, several formal verification algorithms have been proposed to solve the SEC problem. The forward product machine traversal discussed in [10] starts from the initial state of the two circuits under verification and unrolls the circuit for *N* time-frames such that all reachable states could be reached. However, it has been shown by Pixley that *N* could be extremely large [11]. Hence, forward reachable space traversal is not scalable with the increasing sequential depth of the circuits. Exploring structural similarity between two circuits under verification has been shown effective for reducing the CEC complexity [39][40][41], and [30][42] have extended the idea into sequential verification. In [42], the problem is divided into a set of easier sub-problems: verifying candidate flip-flop pairs, internal gate pairs and then output pairs. ATPG and local BDD techniques can be applied to detect equivalent signal pairs, symbolic techniques are then applied to deal with portions of design which are sequentially different. A Bounded Model Checker (BMChecker) and an

Invariant Checker (IChecker) are proposed in [30], both of which are based on circuit SAT techniques to verify internal invariant candidates as well as primary outputs. However, for hard SEC instances that have few structural similarities and different state encodings, there might not be many such constraints to prove the circuits equivalent. To tackle this problem, more general relations among signals need to be learned; techniques such as mining general potential constraints on flip-flops [43], cross time-frame states pair [31] and multi-node invariants among target flip-flop sets [32] have been proposed. Although such approaches could provide more constraints to prune the search space, they also require the engine to examine a huge invariant candidate set, demanding significant space and time resources. In particular, the cost of proving inductive invariants increases quadratically with the increasing size of the candidate set. This is due to the nature of the fixed-point proving process that requires multiple iterations of going through the candidate set. More recently, another novel method was pioneered by [47], [48] implemented and improved it as Property Directed Reachability (PDR). It proposed a new direction for hardware model checking and proved better performance on several instances while eluding both BMC and BDD reachability.

1.4 Contributions of this Thesis

In this thesis, we propose a novel algorithm which reduces the potential simulation-based multi-node invariant set size while maintaining efficiency of the equivalence proof. Compared to previous methods, we treat different kinds of invariant candidates with customized generation algorithms. We first identify constant and equivalent signals among internal signals and flip-flops. Then, we extract general two-node non-equivalent relations from the flip-flop set. Thirdly, we utilize an intelligent algorithm to group flip-flops into smaller subsets for multi-node invariant generation in order to avoid an exponential growth of invariant candidates. Next, we reduce the number of potential multi-node invariants without inadvertently losing pivotal constraints by cleverly merging some of them with a constrained logic synthesis technique. In the past, merging of such potential invariants might lose important relations if the merged relation does not turn out to be a true invariant. Our method mitigates this problem while still reducing the number of potential invariants to a reasonable size and therefore, finishes SEC much faster.

Experimental results show the effectiveness of our approach on SEC under two functional equivalent circuits with completely different state-encodings and few internal equivalences. Techniques that rely only on internal signal equivalences and two-node relations are insufficient to prove such designs. Our approach is up to 20× faster compared to previous mining-based methods for larger circuits.

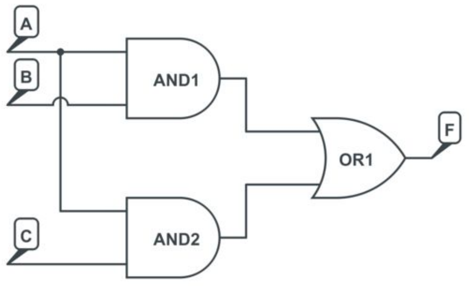
The rest of the thesis is organized as following. Chapter 2 provides background. Chapter 3 details our multi-node simulation-based invariant generation approach. Chapter 4 discusses our constraint logic synthesis technique for such multi-node invariant candidates. Section 5 concludes the thesis and mentions the future work.

Chapter 2

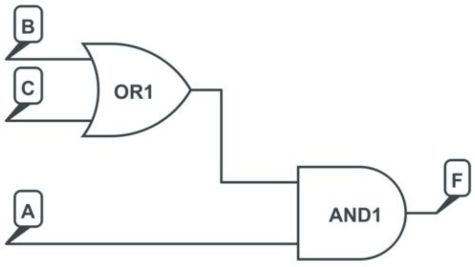
Background

2.1 Equivalence Checking

In general, for a given specification of a circuit, there exists an exponential number of designs that can implement the spec since we may not only encode the Finite State Machine (FSM) differently but also synthesize combinational logic in varying ways. Let's consider a simple combinational design here, *F=AB+AC*, which is shown in Fig. 2.1.a. We could also represent the function as *F=A(B+C)* which is illustrated in Fig. 2.1.b. Obviously, these two designs are structurally different while logically equivalent.



(a) F=AB+AC



(b) F=A(B+C)

Fig. 2.1 Equivalent Circuit Example

2.1.1 Miter Circuit

Equivalence checking attempts to verify if two structurally different designs are functionally equivalent for all reachable states. A miter circuit is constructed from two separate designs being checked as shown in Fig. 2.2. We tie the corresponding inputs together like *IN1* and *IN1'*, so that both circuits receive the same input vector stream. Next, each pair of outputs is fed to an XOR gate and all XOR outputs are fed to a final OR gate. For example, *OUT1* and *OUT1'* are inputs for an XOR gate and *XOR1* and *XOR2* are fed to final output in Fig. 2.2. EC is trying to verify whether the final output is constant 0 or not. SEC, thus, can play a critical role in the design process since the designers might want to try different state encodings and optimization strategies with regard to area, power, or performance.

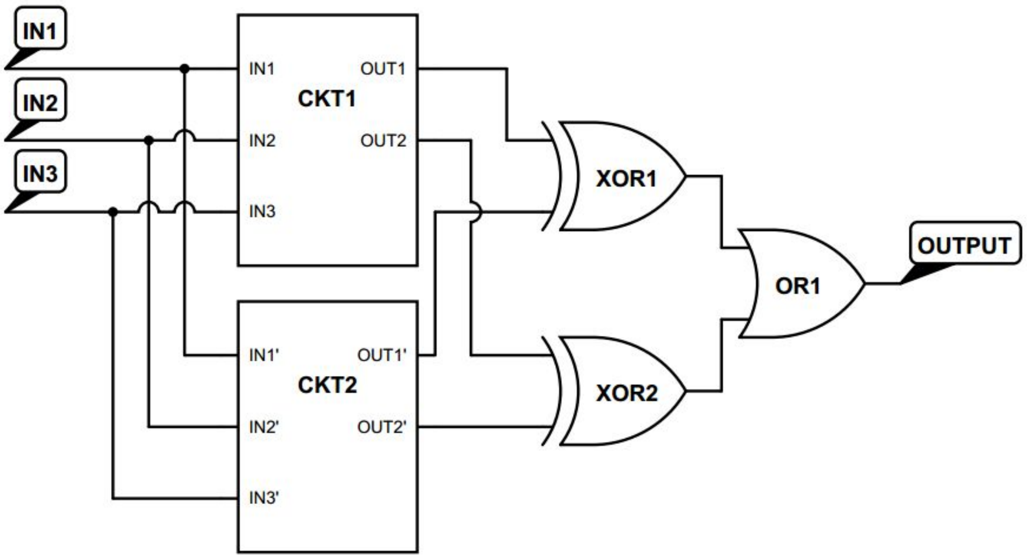


Fig. 2.2 Miter Circuit

2.1.2 Combinational Equivalence Checking

A combinational circuit is a digital logic design where the output is a pure function of current inputs only. There are no state elements or feedback loops in these designs. The construction of combinational logic can be achieved by using AND, OR and NOT gates. However, other gate types such as NAND, NOR, XOR, etc., can be used as well.

Combinational Equivalence Checking (CEC) is a significant component in equivalence checking field, which is to check the output of two combinational designs. Fig. 2.1 is a simple example of two designs under verification for CEC. In the example, if all input vectors for ABC from 000 to 111 have been applied, we can conclude whether the two circuits are equivalent or not based on their output value.

When facing large scale combinational circuits, we may need some advanced algorithms to avoid going through all input vectors. In practice, large CEC instances can be solved efficiently because much research has been done. BDDs have been employed [5][6] to perform functional comparisons for proving the equivalence of internal nets as well as entire circuits. SAT based approaches [7][8] deduce additional knowledge which is added to the CNF to speed the search. Ronald [9] has proposed an effective ATPG based method to reduce the number of false negatives and implement an intelligent cutpoint justification ordering scheme.

2.1.3 Sequential Equivalence Checking

A sequential circuit is a digital logic design whose output depends not only on the present input signal values, but also on the past history of its inputs. In contrast to combinational circuit, sequential circuit result is not a pure function of its input vector, it also has memory elements (flip-flop) or loop to help store the previous state information. In other words, a sequential circuit is combinational logic with state information, which means that we could treat every sequential circuit as two parts, combination logic and memory part. Fig. 2.3 shows a view of a sequential circuit.

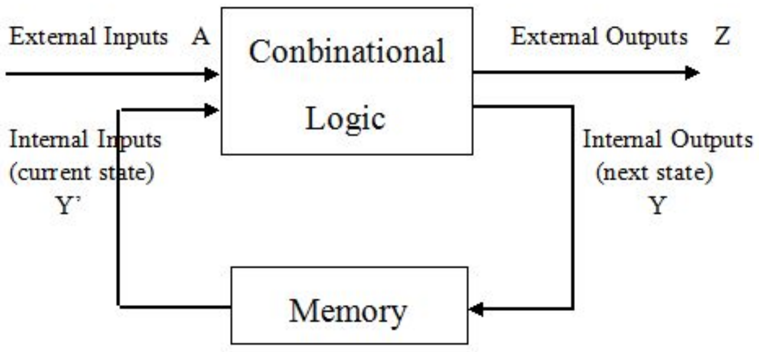

External Inputs A
Conbinational Logic
External Outputs Z
Internal Inputs (current state) Y'
Internal Outputs (next state) Y
Memory

Fig. 2.3 Sequential Design

A sequential logic can be defined as: (*S, S_0, I, O, δ, λ*), where *S* is the reachable state set, S_0 is the initial state, *I* is the set of external input vectors, *O* is the set of external output vectors, *δ* is the next state function ($I \times S \rightarrow S$) and *λ* is the output function ($I \times S \rightarrow O$). Obviously, for a given specification of a circuit, there exists an endless number of designs that can implement the spec since we may not only encode the Finite State Machine (FSM) differently but also synthesize combinational logic in varying ways.

Sequential Equivalence Checking (SEC) is to compare two sequential designs are functional equivalent or not based on same specification. Two sequential designs could be equivalent if they keep corresponding primary outputs equivalent while going through the whole reachable state space.

The number of flip-flops of two designs, however, does not need to keep the same. This can cause two functional equivalent designs look very different from each other. The use of SEC has become prevalent in the design and verification process since the designers might want to try different state encodings and optimization strategies with regard to area, power, or performance.

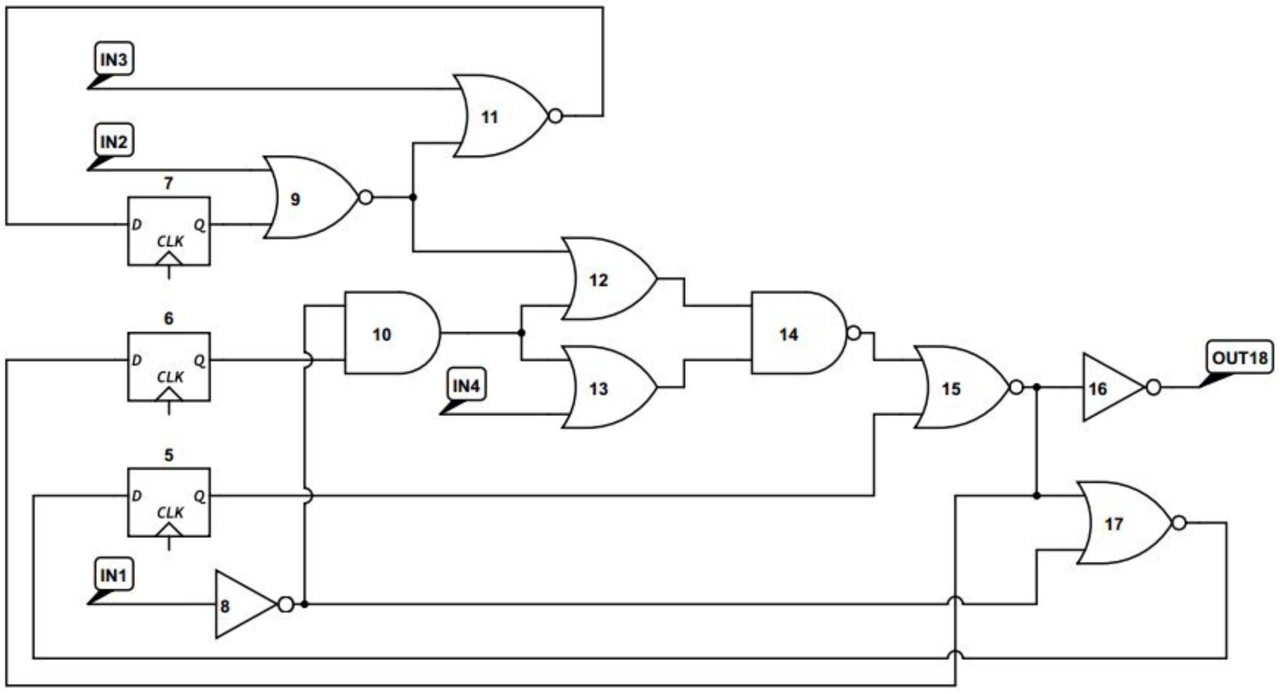
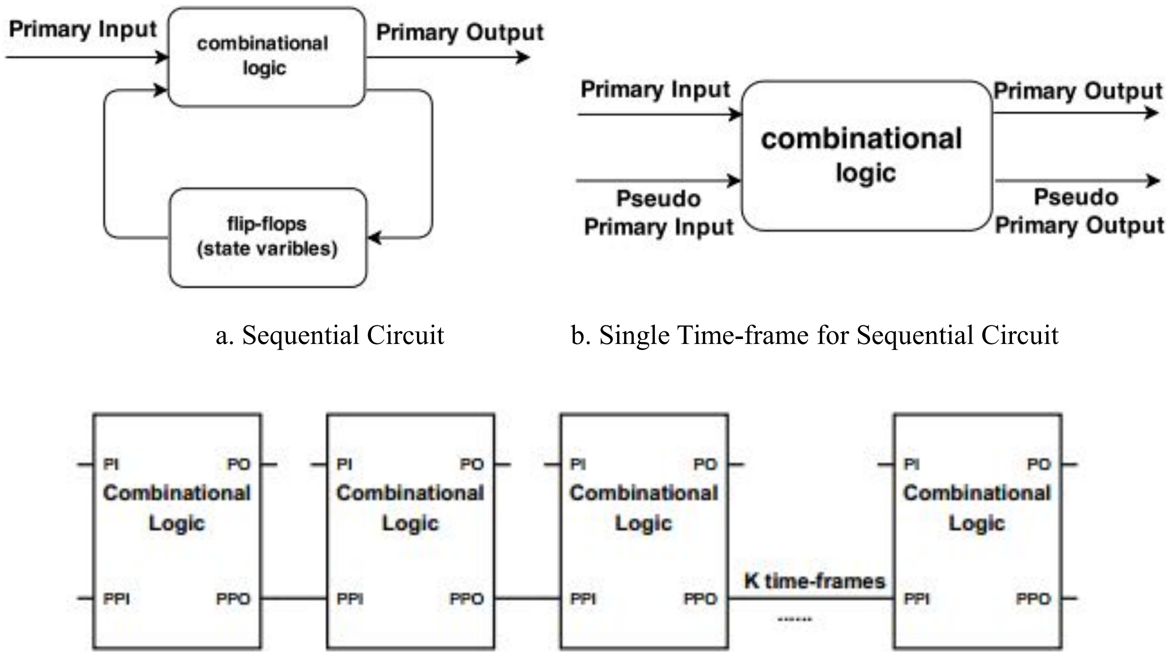


Fig. 2.4 Sequential Circuit

Unlike CEC which one just needs to traversal input vector set, SEC also requires one to consider the reachable state set, which is more challenging. The number of possible states in a sequential circuit is 2^N with N being the number of flip-flops. Obviously, the number of states is exponential with respect to the number of flip-flops. On the other hand, only a portion in this 2^N state set might be reachable based on the design structure, which is also hard to compute for large designs. Let's explain it with example shown in Fig. 2.4. We know that gate 10 set to logic 1 leads gate 17 to logic 0 in the previous time frame, and gate 17 set to logic 0 leads gate 5 to logic 0 in the next time frame ($10 \xrightarrow{-1} \overline{17} \xrightarrow{1} \overline{5}$). And when gate 10 is logic 1, both gates 6 and gate 8 need to be set to logic 1 in the same time frame based on AND gate property. Thus we will know state 110 (on flip-flops 5, 6, 7) is an unreachable state when input 1 is logic 0 for this sequential design.

In this thesis, the basic idea for SEC is to convert the circuit into a combinational one by unrolling the design into several time-frames and then prove it by adding sufficient simulation-based inductive invariants to prune search space. There are two basic ways to represent the circuit: BDDs and Conjunctive Normal Form Satisfiability (SAT). We will discuss this part in detail later.

2.2 Unrolling Circuit



a. Sequential Circuit b. Single Time-frame for Sequential Circuit

c. Unroll Circuit

Fig.2.5 Unroll Sequential Circuit

For a sequential circuit, we need to convert it into combinational logic by unrolling the design. We know from previous sections that we could treat the sequential logic as two parts, combinational part and memory elements which is shown in Fig. 2.5.a. To unroll it, we have to open the loop in memory elements, which means for each flip-flop, we convert it into two corresponding signals: a Pseudo Primary Input (PPI) and a Pseudo Primary Output (PPO) demonstrated in Fig. 2.5.b. For now, we have already convert sequential circuit into combinational logic, in which there is no loop or memory elements. Then we should connect every time frame together to build *K* time-frame unrolling circuit as shown in Fig. 2.5.c. Since for each state, we need to calculate its value using the current input and previous state information ($I \times S \longrightarrow S$), we then connect the next time-frame PPI with current time-frame PPO, which enable the signals to propagate values from one time-frame to the next one. The first time-frame here is called initial time-frame. We could know from Fig. 2.5.c that its PPIs are totally controllable. Therefore, we may assign an illegal assignment for them. To avoid such problem, we should set them into an initial state (*S_0*). If there is no initial state for some designs, we should random simulate the design into a completely specified state (no flip-flop holds unknown value) and use it as the initial state.

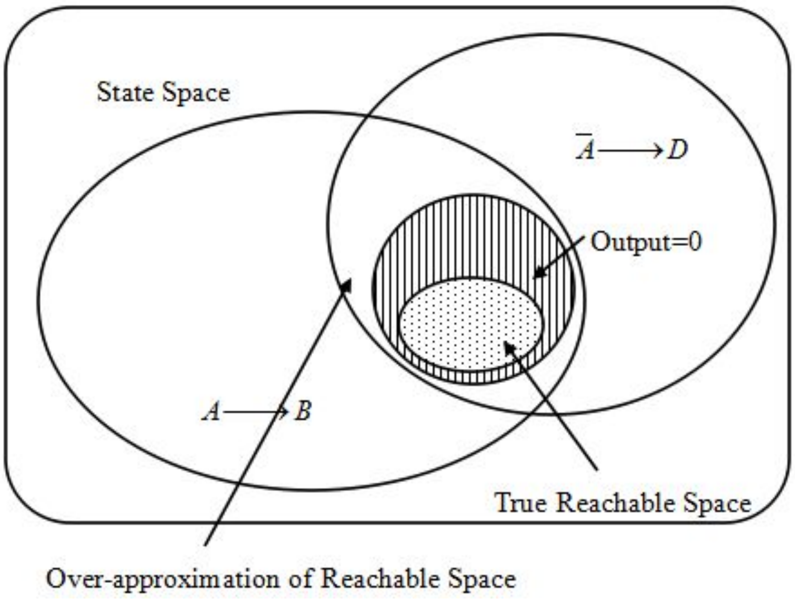
A two time-frame unrolling circuit model is the base of two time-frame assume-then-verify model which is used for proving inductive invariants, and the multi time-frame unrolling model is used in this thesis for our filtering algorithm, which will be discussed later.

2.3 Implications and Invariants

A relationship among signals in a circuit is called an invariant, also known as implication. Implications have been used in many Electronic Design Automation (EDA) applications, such as testing [51, 52, 53], post-silicon validation [54, 55], bounded model checking [56], and hardware Trojan detection [57]. For SEC, we could translate such invariants into Conjunctive Normal Form (CNF) and then treat them as constraints to shrink our search space. A sufficient number of true invariants will effectively restrict the search space such that target properties can be proven, which may help sequential design equivalence checking.

There are two categories of invariants. The first is static invariants, which hold true in the entire state space, reachable or otherwise. Static invariants can be computed by analyzing the logic structure of the circuit. The basic idea is to set a gate to a logic value and then propagate this value as much as possible until no new gate value could be implied. Much work has been conducted for static implications over the years and some of this work could be found in [12][13]. By pre-computing static implications, in some BMC cases, more than 100× speedup can be achieved over the conventional BMC on several difficult safety properties [14]. Longer unrolled BMC instances can offer more opportunities for speedup as static invariants can help to prune larger search spaces.

The other category is inductive invariants. In contrast to static implications that hold in all states, inductive invariants hold in the reachable states but may or may not hold in unreachable states. Thus, they can be used to prune away unreachable states as shown in Fig. 2.6. In this figure, two inductive invariants $A \rightarrow B$ and $A' \rightarrow D$ hold in all reachable states, but they do not hold in some unreachable states. Thus any state that violates either of these two invariants can be deduced as unreachable. Furthermore, we can conclude that any reachable state must hold both invariants. By calculating the intersection of the states constrained by each invariant, we obtain a much smaller over approximate reachable space for SEC. This over approximate reachable space can often be further reduced by adding more inductive constraints. If the pruned state space makes it impossible for the miter output to be set to a logic 1 (equal to or smaller than the shaded ellipse as shown in figure), we can conclude that the two circuits are equivalent.



State Space
$\overline{A} \longrightarrow D$
Output=0
$A \longrightarrow B$
True Reachable Space
Over-approximation of Reachable Space

Fig. 2.6 Prune Reachable Space by Invariants

2.3.1 Direct Implications

Direct implications are the fundamental relations based on on the property of a Boolean gate. There are two types of direct implications: direct forward implication and direct backward implication.

For a direct forward implication, the control value of gate is needed. Let's use an OR gate as example to illustrate this concept. Signal A and B are inputs to our OR gate and signal C is the output here. Assume signal A is set to logic 1, A=1 will be propagated to signal C no matter which value is applied to signal B based on the OR gate property: ($A(1) \longrightarrow C(1)$). Therefore, 1 is the control input value for OR gate. The control values of 4 basic gate types are shown in Table 2.1. For BUF and NOT gate, their outputs are always changing together with the input values, we could also add such relations into our direct implication set. Direct backward implications are the contra-positive relations of the direct forward implications. The term contra-positive law means that if

$A(w) \longrightarrow B(v)$, then $B(\overline{v}) \longrightarrow A(\overline{w})$. This law enables the algorithm to discover unilateral indirect implications [16]. We could also use the previous OR gate for example. We could get direct forward implications $A(1) \longrightarrow C(1)$ and $B(1) \longrightarrow C(1)$, according to contra-positive law, we could also get $C(0) \longrightarrow A(0)$ and $C(0) \longrightarrow B(0)$. Thus for an OR gate, the output is logic 0 will implies both inputs to logic 0. We could get such relations easily for AND, NAND and NOR gate.

Table 2.1 Control Value Table

Gate Type	Controlling Value	Non-controlling Value
AND	0	1
NAND	0	1
OR	1	0
NOR	1	0

2.3.2 Indirect Implications

Indirect implications capture some non-trival relations in the circuit which direct implications fail to detect. To get indirect implication relations, we may first assert the target gate with logic 1 or 0. Then we perform direct implication method introduced before and apply all the implied gate values to run circuit simulation. As a result, more gates may change their logic value from unknown state to known state (logic 1 or 0). We say such new relations are indirect implications.
The formula to describe indirect implication is:

$S(indirect\_impl) = S(direct\_impl) \cup simulation(S(direct\_impl))$.

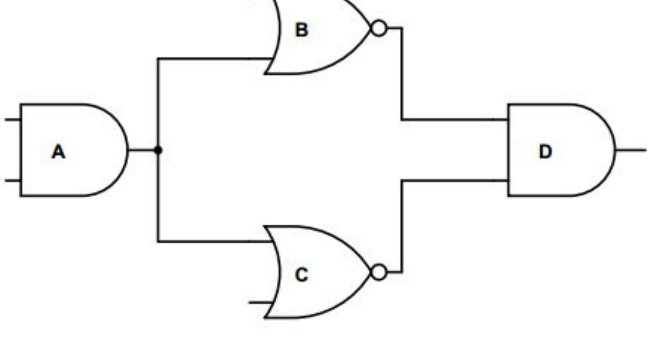


Fig. 2.7 Indirect Implication

Here is an example in Fig. 2.7. When we assert logic 1 to gate A, we could get both logic 0 at gate B and C since logic 1 is the control value for the NOR gate. Then we will propagate logic 0 to gate D when we pass all such known gate values to simulation step, that relation could be represented as $A(1) \longrightarrow D(0)$. While, we could not extract such relation by direct implication only since it may just consider single gate and ignore the whole structure. If the design is complex enough or we may say if there are lots of split-then-merge structure (here in Fig. 2.7, gate A is input to both gate B and C, these two signals are merged together as input to gate D later), indirect implication will bring us many valuable relations. Note that we could also use BCP technology computing indirect implications.

2.3.3 Extended Backward Implications

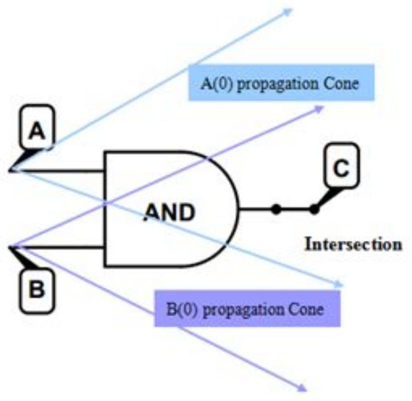
Extended backward implications attempt to discover additional indirect implications which previous methods fail to detect at low cost. Zhao [12] first proposed this new static logic implication algorithm. Finding lots of such relations among circuit signals has many useful applications, one of which is to build more powerful filters for SEC. The basic idea is to consider unjustified gates in the implication set. An unjustified gate means that the gate value is specified but none of its inputs value is specified. Generally, we could apply extended backward implication on AND, NAND, OR, NOR, XOR and NXOR gate types. Here, we use AND, OR and XOR types as examples.

AND gate

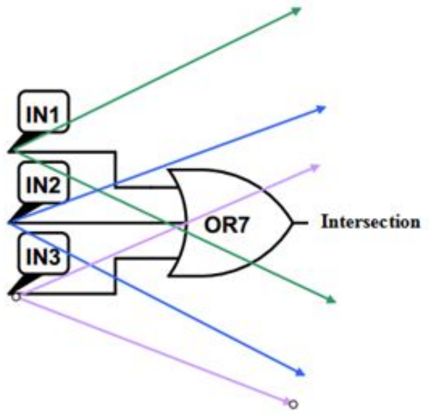
For an AND gate (Fig. 2.8.a), if the output value is logic 1, we could get $C(1) \longrightarrow A(1)$ as well as $C(1) \longrightarrow B(1)$ according to previous direct backward implications. However, there is no certain backward implication rules to apply when the output is logic 0 since there are multiple input combinations that could propagate logic 0 at gate C (AB=01, 10 or 00). The basic idea of extended backward implications is calculating the intersection set of all possible situations. If the output is logic 0, we could apply algorithm:

$$Ex\_bk\_impl(output,0) = \bigcap_{i=1}^{m} simulate(impl(output,0) \bigcup impl(input_i,0))$$

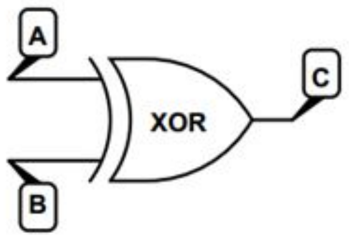
The formulation above means that if gate C is logic 0, either input A or input B should be logic 0. If input A is logic 0, we could take the union of implication set of C(0) and implication set of A(0) together, then run circuit simulation based on the above relations. Here we could get A(0) propagation set shown in Fig. 2.8.a. We could apply exactly the same algorithm on input B. Since if we want to produce logic 0 at AND gate output, at least one of its inputs need to keep logic 0, we then calculate the intersection of all propagation sets as extended backward implication set for AND type gate.



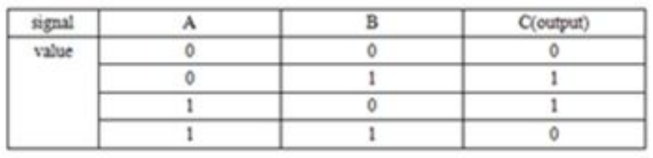
a



b



c



signal	A	B	C(output)
value	0	0	0
	0	1	1
	1	0	1
	1	1	0

d

Fig. 2.8 Extended Backward Implication

OR gate

If there is an OR gate with output being logic 0, we could get $output(0) \longrightarrow input_i(0)$ according to previous direct backward implications. Again, there is no certain backward implication rules to apply when the output is logic 1 since there are multiple input combinations that could lead output=1 (01, 10 or 11). Then extended backward implication is an extension of that:

$$Ex\_bk\_impl(output,1) = \bigcap_{i=1}^{m} simulate(impl(output,1) \bigcup impl(input_i,1))$$

Let's explain this equation using example in Fig. 2.8.b. From the OR gate property, any logic 1 input could lead output value to logic 1, logic 1 is the control value for OR gate. Therefore, if we get logic 1 at OR gate output first, we may deduce that at least one of its input values need to be logic 1, we calculate intersection of all implication sets as final result (intersection part shown in Fig. 2.8.b). For each implication set, we could also take the union of implication set of output(1) and implication set of one input(1) together, then run circuit simulation based on the above relations. NOR gate will share similar equation here.

XOR gate (2 inputs)

Fig. 2.8.c is an XOR gate example and Fig. 2.8.4 is the truth table of it. There is no control value in XOR type gate which means we could not apply any kind of direct implications, therefore we may consider every case separately and finally merge them together.

When output gate C is logic 0, there still exists 2 conditions, both inputs are logic 0 or both inputs are logic 1 based on the property of XOR gate. Therefore, we could take intersection of these two implication sets to make sure our output is const 0. For each condition, we could apply all three relations. The equation is shown here:

$$Ex\_bk\_impl(output,0) = simulate(impl(output,0) \bigcup impl(input_1,0) \bigcup impl(input_2,0)) \\ \bigcap simulate(impl(output,0) \bigcup impl(input_1,1) \bigcup impl(input_2,1))$$

When output gate C is logic 1, we could apply exactly the same algorithm here. Note that when the inputs hold the different values from each other, we could propagate logic 1 at output. Therefore, we could get the following equation:

$$Ex\_bk\_impl(output,1) = simulate(impl(output,1) \bigcup impl(input_1,1) \bigcup impl(input_2,0)) \\ \bigcap simulate(impl(output,1) \bigcup impl(input_1,0) \bigcup impl(input_2,1))$$

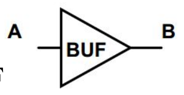
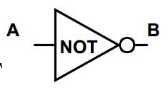
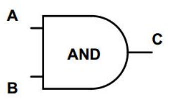
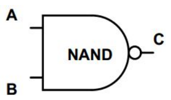
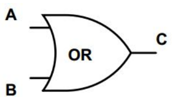
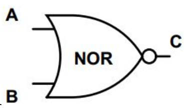
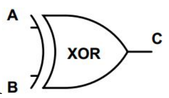
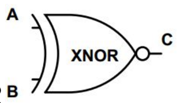
2.3.4 Inductive Invariants

Inductive invariants are gate relations that hold in all reachable states but may or may not in unreachable states. For SEC, inductive invariants may form powerful constraints to prune the search space especially when the system contains huge don't-care spaces as shown in Fig. 2.6.

Unlike static invariants, inductive invariants cannot simply be learned directly from the circuit structure; instead, they need to be inductively proved. To compute such inductive invariants, we need to first identify a set of potential invariants. This can be done by mining (or searching for) relations from a simulation database. That is, the database contains the values of circuit signals from simulating a vector set. If a relation holds in the entire database, we record it as a potential invariant candidate and will need to prove it later on. For example, if two signals *x* and *y* never have combined values of 01 in the database, then we mark (*x* + *y'*) as an invariant candidate. We then call invariant checker to prove its correctness.

2.4 CNF

Table. 2.2 CNF Table

Gate Type	CNF
 BUF	$(\overline{A} \cup B) \cap (A \cup \overline{B})$
 NOT	$(\overline{A} \cup \overline{B}) \cap (A \cup B)$
 AND	$(A \cup \overline{C}) \cap (B \cup \overline{C}) \cap (\overline{A} \cup \overline{B} \cup C)$
 NAND	$(A \cup C) \cap (B \cup C) \cap (\overline{A} \cup \overline{B} \cup \overline{C})$
 OR	$(\overline{A} \cup C) \cap (\overline{B} \cup C) \cap (A \cup B \cup \overline{C})$
 NOR	$(\overline{A} \cup \overline{C}) \cap (\overline{B} \cup \overline{C}) \cap (A \cup B \cup C)$
 XOR	$(\overline{C} \cup A \cup B) \cap (\overline{C} \cup \overline{A} \cup \overline{B}) \cap (C \cup A \cup \overline{B}) \cap (C \cup \overline{A} \cup B)$
 XNOR	$(\overline{C} \cup A \cup \overline{B}) \cap (\overline{C} \cup A \cup \overline{B}) \cap (C \cup A \cup B) \cap (C \cup \overline{A} \cup \overline{B})$

In Boolean logic, Conjunctive Normal Form (CNF) is a conjunction of clauses ($clause1 \cap clause2 \cap \ldots$), where a clause is a dis-junction of literals ($literal1 \cup literal2 \ldots$). For each literal, it can be represented either a positive polarity (A) or negative polarity ($\overline{A}$) of a signal. SAT Solver typically requires the input to be in CNF formula and attempts to find an assignment to the variables such that all Boolean relations are satisfied. Therefore, we need to translate the problem under verification into CNF. To prove SEC in particular, we need to convert miter circuit into CNF formula as well as assert final output constant logic 1 in SAT Solver.

Every logic circuit can be converted into its equivalent CNF formula. This transformation is based on logical equivalences rules. Table. 2.2 illustrates the convention between different types of gates and the corresponding clauses. Let's take the buffer gate as an example. The input and output of buffer should always be the same, according to truth table of its corresponding CNF formula, we will notice they are logic equivalence all the time.

2.5 SAT Solver

SAT Solver has a wide range of applications in VLSI design, including the design verification area and the sequential equivalence checking problem. The basic approach is (A), converting the whole miter circuit into CNF formula as input to Solver, (B), asserting final miter circuit output to logic 1 as an additional clause, (C), if the SAT solver could find a solution satisfying our input constraints in the reachable state set, we may conclude that there exist an assignment which leads two design functionally different and (D), otherwise, if there is no solution in all search space, we could prove the two designs are functionally equivalent.

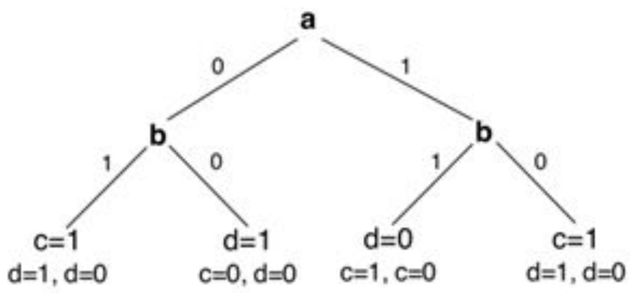
The problem of determining the satisfiability is the first proven NP-complete problem by Stenpen Cook and Richard Karp [28][29]. There is no known algorithm that efficiently solves SAT and it is generally believed that no polynomial-time algorithm exists. However, a large amount of research has been done trying to develop more efficient search algorithms, resulting in modern SAT solvers which could handle million level variables and clauses. A survey on recent advanced SAT solver can be found in[18]. We will discuss some efficient search algorithms used in advanced solver in the following part.

2.5.1 Backtracking Search Based Algorithm (DPLL)

In 1960s, Davis, Putnam, Logemann and Loveland proposed a SAT search algorithm [19][20], DPLL. After almost 50 years, DPLL algorithm still forms the basis for most efficient SAT solvers. It's a chronological backtracking algorithm. Let's explain the procedure using CNF example in Fig. 2.9.a. We first need to check pure literals. If there are some in our clause set, we could save literal value and remove such clauses. Here in the example, there is no pure literal. We then should find unit clauses. If there exist such clauses, we should save literal value, remove clause and repeat this process until there is no new unit clause. Thirdly, we could pick a free variable and set it to either true or false which is called making a decision, we must record all decisions we made before in order. Next, based on the decision we made before, we could repeat the previous two steps to set down more literal values. In most cases, it may deduce a conflict. Here in Fig. 2.9.b, we need d=1 and d=0 at the same time when we set ab to 01. Now backtracking is necessary by reversing the last decision (b = 1) and continuing. Finally, we could keep going until either we find a solution or we traversal all space and prove there is no assignment here. The whole algorithm is shown in Fig. 2.9.c. The time complexity is 2^n in worst case since we may have to make decisions in every literal twice(backtrack every decision).

$$(\bar{a} \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee c) \wedge (\bar{c} \vee d) \wedge (\bar{c} \vee \bar{d})$$
$$\wedge (\bar{a} \vee c \vee d) \wedge (\bar{a} \vee b \vee \bar{d}) \wedge (b \vee c \vee \bar{d}) \wedge (a \vee b \vee d)$$

(a). DPLL Clauses



(b). DPLL Search Tree

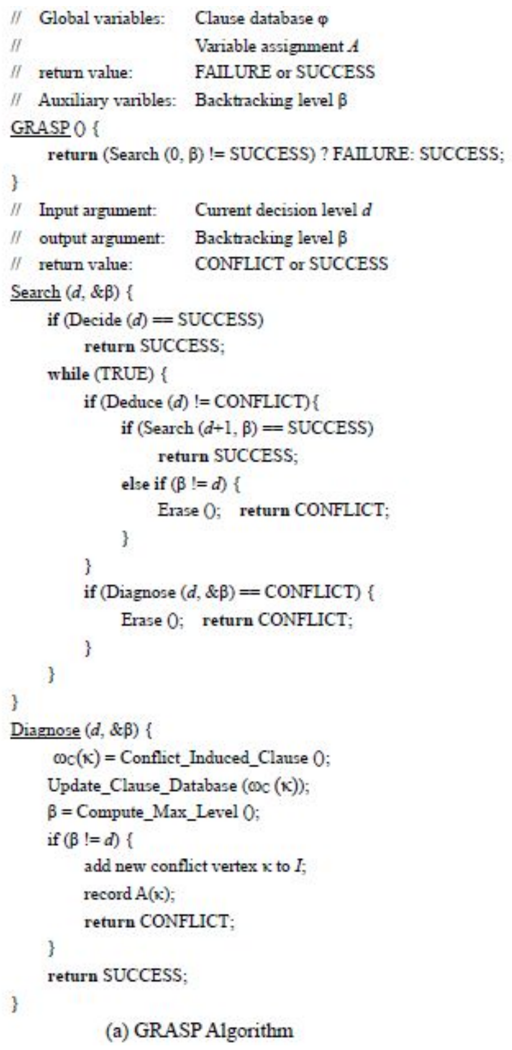
```
DPLL() {
   while (ChooseNextAssignment()) {
      while (Deduce () == CONFLICT) {
         blevel = AnalyzeConflict () ;
         if (blevel < 0) return UNSATISFIABLE;
         else BackTrack (blevel);
      }
   }
   return SATISFIABLE;
}
```

(c). DPLL Algorithm

Fig. 2.9 DPLL Example [4], C. Baier and J.P. Katoen. *Principles of model checking*. Vol. 26202649. Cambridge: MIT press, 2008. Used under fair use, 2015

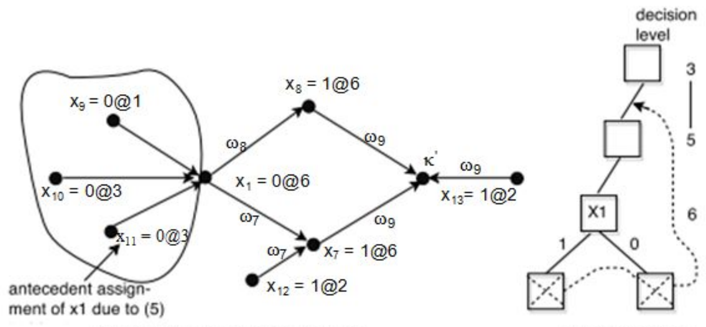
2.5.2 GRASP

SAT solver could not be applied into practical field for several decades since DPLL has been proposed because it’s time complexity. However, Marques-Silva and Sakallah proposed an improved backtracking search algorithm [21] in 1997 which could speed up search procedure dramatically.



```
// Global variables:     Clause database φ
//                       Variable assignment A
// return value:         FAILURE or SUCCESS
// Auxiliary varibles:   Backtracking level β
GRASP () {
    return (Search (0, β) != SUCCESS) ? FAILURE: SUCCESS;
}
// Input argument:       Current decision level d
// output argument:      Backtracking level β
// return value:         CONFLICT or SUCCESS
Search (d, &β) {
    if (Decide (d) == SUCCESS)
        return SUCCESS;
    while (TRUE) {
        if (Deduce (d) != CONFLICT){
            if (Search (d+1, β) == SUCCESS)
                return SUCCESS;
            else if (β != d) {
                Erase (); return CONFLICT;
            }
        }
        if (Diagnose (d, &β) == CONFLICT) {
            Erase (); return CONFLICT;
        }
    }
}
Diagnose (d, &β) {
    ωC(κ) = Conflict_Induced_Clause ();
    Update_Clause_Database (ωC (κ));
    β = Compute_Max_Level ();
    if (β != d) {
        add new conflict vertex κ to I;
        record A(κ);
        return CONFLICT;
    }
    return SUCCESS;
}
```

(a) GRASP Algorithm



(b) Conflicting implication sequence (c) Decision tree

Fig. 2.10 GRASP Learning Process [21], J. P. M. Silv and K. A. Sakallah. "GRASP—a new search algorithm for satisfiability." In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 220-227. IEEE Computer Society, 1997. Used under fair use, 2015

GRASP is very similar to the DPLL algorithm in structure, however, it could learn some information whenever we analyze conflicts (shown in Fig. 2.10.a as Diagnose part in GRASP flow). We call such information learned clauses and we add them to our original clause set to avoid searching wrong path in the future. Thus, we could change our search process from chronological to non-chronological which makes the SAT solver more practical. In order to learn relations, we need to build implication graph when searching the space. When a conflict arises during BCP, the implication graph converging on the conflict node is analyzed to determine those (unsatisfying) variable assignments that are directly responsible for the conflict. Negation of this implicant, therefore, yields some learnt clauses that is not contained in original clause set. Those new information will help our search engine avoiding traversal some paths which is shown in Fig. 2.10.b. Therefore in decision tree shown in Fig. 2.10.c, we may go back to decision about variable 3 instead of variable 5 using relations we learned from left graph.

2.5.3 Fast Deduction

SAT solver could handle millions of variables when we introduced backtracking algorithm and conflict analysis procedure. It's still far away from the need in the real world. And we found that 80% of solver search time is spent on Boolean propagation part. The organization of the clause database affects the propagation speed and is therefore important. In 2000, Moskewicz and Matthew introduced fast deduction algorithm[22]. Chaff SAT solver, which is based on this algorithm, could handle more variables.

They proposed a new kind of data structure storing literal and clause information which is called 2-literal watch list for each clause. For each literal, a list is also maintained with the clauses in which that literal is currently watched. Then, after every decision, they could simply scan the specified literal list and lazily update the two watch pointers in every clause to check the clause status: satisfied, conflict or assert. When backtracking taking place, the watched literal lists do not need to be modified. This allows very efficient backtracking.

To sum up, a large amount of time and effort has been spent in developing novel SAT Solver search algorithms which has led a huge improvement in solver performance. The state-of-art SAT solvers could handle million level variables and even more clauses nowadays. Zchaff is a improved version of chaff solver and it's reported to have success stories to solve problems with more than one million variables and 10 million clauses, it could also support incremental solving which will be quiet useful in our research. MiniSat [23][24][25] provides a small, yet efficient solver which core code is less than one thousand lines and also could beat most recent solvers. It's easy to read and modify and therefore could be used for integration for a bunch of EDA applications. Glucose[36], based on Minisat and proposed since 2010, has become the solver star since it predicts learnt clauses successfully.

Chapter 3

Efficient Simulation-based Multi-node Invariants Generation and Pruning Framework I

In this chapter, we describe our simulation-based multi-node invariant generation and pruning framework. For hard SEC instances that have few structural similarities and different state encodings, we note that constant and equivalence invariants are insufficient to prove the circuits equivalent. Thus, general relations among signals are needed to provide additional constraints to prune the search space. However, there is an ocean of multi-node invariants, which will require an enormous computational cost to find and prove. To address these concerns, we propose our simulation-based multi-node invariant generation and pruning scheme which reduces the invariant candidate set size while maintaining the proof efficiency. Our main interest is focusing on flip-flops, which should be more valuable compared to general internal nodes in miter circuit. On the other hand, single- or 2-nodes invariants could impose more powerful constraints on the state space than multi-node invariants. To take advantage of this property, we address this problem utilizing a clever potential invariants generation scheme: A, We unroll the sequential circuit multiple times and use BCP to extract static implications containing flip-flops as preparation for further SAT search. B, we treat different kinds of invariant candidates with customized generation algorithms. We first identify constant and equivalent signals among internal signals and flip-flops. Then, for general multi-node invariants, in order to avoid an exponential growth of invariant candidates, we utilize an intelligent algorithm to group flip-flops into smaller subsets for invariant candidate generation. Experimental results show that we could reach conclusion much faster than the mining-based approach mentioned in [15][32].

The rest of the chapter is organized as follows. Section 3.1 is our motivation. Section 3.2 introduces how we use random simulation to derive potential invariants for single- and two-nodes. Section 3.3 indicates BCP static implication among multiple time-frames and its application. Section 3.4 presents details for assume-then-verify invariants checker. Section 3.5 introduces different kind of filters. Section 3.6 illustrates Bounded Flip-flop Cone of Influence (BFCOI) and Multi-node Missing Pattern Invariants. Section 3.7 describes the SEC framework that we propose. Section 3.8 presents experimental results and conclusion is followed in Section 3.9.

3.1 Motivation

From previous chapter, we know that for SEC, we need to break the loop and convert the design into the full-scan mode which means we treat flip-flops as pseudo primary input (PPI). Thus the SAT solver may assign a combination on PPI such that this assignment is not in reachable state set. In other words, the SAT solver could assign a spurious state on the PPIs. In order to eliminate such false counter example, we need to add sufficient constraints to prune the search space. This chapter comes up with a potential simulation-based multi-node invariant generation scheme which could balance the set size of potential invariants and proof efficiency.

One approach here is to take advantage of original sequential design first. If we unroll sequential circuit into multiple time-frames, we could keep a large amount of internal nodes relations which may not hold in single time-frame. For example in Fig. 2.4, we will get $(10 \xrightarrow{-1} 17)$ if we unroll the circuit into 2 time-frames instead of 1. Obviously we could obtain much more implications when we unroll the design multiple times, which will describe the circuit more accurate (with such implications, our loop-free model is more similar to the original design). However, at the same time, we may face a huge number of redundant relations which may slow down our SAT search procedure. Therefore, we just address flip-flop related implications. Table 3.1 shows different flip-flop implication numbers when we unroll our circuits into 3 and 7 time-frames (we ignore the relations gotten from single time-frame).

Table 3.1 Implication Numbers under Different Time-frames

circuit	# BCP clause (3 TF)	# BCP clause (7 TF)
T1	1,122	1,420
T3	4,398	12,604
T4	5,279	14,735

Second approach here is to take advantage of our new invariant generation scheme. Note that single- or 2-node invariant has more constraining power. To demonstrate this idea, we could put an example here with 10 flip-flops. A 2-node relation on the flop-flop could potentially eliminate 256 states (2 nodes being specified) while 3 nodes and 4 nodes relations could potentially eliminate 128 and 64 states respectively. Thus we first identify constant and equivalent signals among internal signals and flip-flops. Such constraints may still be insufficient when two circuits under verification have different state encodings and very few equivalent signals between them, we then shift our focus to multi-node invariants phase. The invariants of interest here are just flip-flops to avoid explosion issue, however, we may have a huge number of potential invariant candidates which requires significant space and time resources. Just an simple example, for a design with 300 flip-flops, the total number of potential 3-node relations is about 4.45 millions and for 4-node relations is about 3.31 billions. Therefore, we utilize an intelligent algorithm to group flip-flops into smaller subsets for invariant candidate generation, in which we take advantage of the cone of influence under false invariants provided from early stage. Those false invariants may be caused by illegal assignment on flip-flops. If the multi-node invariants describing missing patterns on such flip-flop cone are true, they could help the SAT Solver to avoid such illegal assignment and may lead to sequential designs equivalence conclusion.

3.2 Random Simulation and Invariant Candidate Generation

Table 3.2 Simulation Database

Vector ID	Gate 1	Gate 2	...	Gate m
Vec 1	0	1	...	0
Vec 2	0	0	...	1
...	...	...	...	...
Vec n	1	1	...	0

In this step, we simulate the circuit with a bunch of random input vectors as our invariant candidate generation database. Then we analyze our database to get potential invariant set for further verification.

First of all, the circuit need to be simulated to a fully specified state which means no flip-flop's value is *X*. Then we need to record all gate value information during random vector simulation. After each round, a single bit logic value (0 or 1) is obtained for every gate in the circuit, we record it into our database. Table 3.2 is a example of such database. It's a two dimensional form in which column represents for gate ID and row represents for vector ID. Generally, the more random vector are applied, the more accuracy we will get for candidates. In this thesis, we use a range instead of a fixed number for our input vector number under different instances which is from 3k to 30k.

Data-mining into our database to generate candidates is just looping through all vectors for all internal nodes. If the gate property holds in the whole database, we say it's a candidate and record it for further use. Table 3.3 lists all of the gate properties (for single- and two-node only). For instance, if we want to check $g10 \longrightarrow g20$, we need to check whether relation $\overline{g10} \vee g20$ holds or not for all input vectors. If yes, we tag it as a potential invariants.

Table 3.3 Candidate Property

Gate Property	Containing Gates	Gate Relation
Const 0	g1	$sig(g1) = 0$
Const 1	g1	$sig(g1) = 1$
Equivalent Gates	g1, g2	$sig(g1) = sig(g2)$
Counter Gates	g1, g2	$sig(g1) = sig(\overline{g2})$
$g1 \longrightarrow g2$	g1, g2	$sig(\overline{g1}) \vee sig(g2)$
$g1 \longrightarrow \overline{g2}$	g1, g2	$sig(\overline{g1}) \vee sig(\overline{g2})$
$\overline{g1} \longrightarrow g2$	g1, g2	$sig(g1) \vee sig(g2)$
$\overline{g1} \longrightarrow \overline{g2}$	g1, g2	$sig(g1) \vee sig(\overline{g2})$

3.3 Boolean Constraint Propagation (BCP) Algorithm

BCP or Unit Propagation (UP) algorithm is a procedure that could simplify a set of clauses. It's a core step in SAT solver calculation and we could also use it to get static implication.

First of all, we convert our circuit into CNF. We then unroll the sequential design into several time-frames to obtain more implications from the original structure. For example circuit T4 in Table 3.1, if we unroll the circuit into 3 time-frames, we will get 5,279 BCP clauses involving flip-flops compared to 14,735 new clauses if we unroll it into 7 time-frames. Next, we inject an unit clause into our clause set. Assume we want to calculate the implication set when gate m is constant 0, we add $(\overline{m})$. From here, we go through all the clauses in our set following two rules: A, remove clause that contains $\overline{m}$. B, in every clause that contains m, delete such literal in clause. If the new clause is an unit clause n, we could get an implication $(\overline{m} \longrightarrow n)$ and repeat the previous step using new literal clause. Otherwise, if the new clause is empty, we will face conflict here which means gate m should be logic 1.

If we apply BCP for all gates in our design, we could easily get huge number of implications that might slow down our SAT search step. Thus we just focus on the implications that involve flip-flops in this chapter, trying to shrink the reachable state set size. For example in Fig. 2.4, we could get implication $6 \longrightarrow \overline{5}$ which means state 11x is unreachable from BCP.

3.4 SAT-based Inductive Invariant Checker

Mathematical induction is a method of proving mathematical results: An assertion $A(x)$, depending on a natural number x, is regarded as proved if $A(1)$ has been proved and if for any number n, the assumption that $A(n)$ is true implies that $A(n+1)$ is also true [33]. We could also apply the principle into bounded model checking field.

Bounded Model Checking (BMC) with induction was introduced in [34][35]. Later, Lu proposed a SEC framework based on K^{th} invariant[30]. The advantages of such invariants are that with a small unrolling and without computing all reached states, we could possibly prove the target invariant.

As the name suggests, both a base case and an inductive step are needed to prove inductive invariants. A single time-frame of the circuit is used for the base case, in which the Pseudo Primary Inputs (PPI) should be constrained to a known valid initial state S_0 (this state can be obtained by taking the first initialized state from random simulation). The potential invariant to be checked is then added to this single-frame circuit to complete the base case setup as shown in Fig. 3.1. Every potential invariant is checked one at a time to determine whether it passes the base case or not.

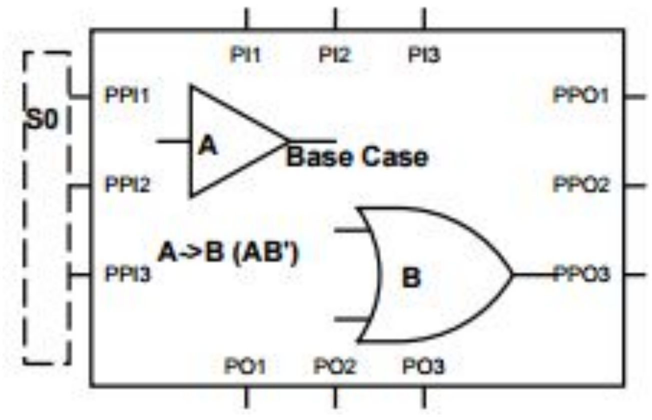


Fig. 3.1 Base Case Framework

Table 3.4 Conversion Form

Relation Type	CNF	Objective	
$A = 1$	A	$\overline{A}$	
$A = 0$	$\overline{A}$	A	
$A = B$	$(A + \overline{B})(\overline{A} + B)$	$C = A \oplus B$	C
$A = \overline{B}$	$(A + B)(\overline{A} + \overline{B})$	$C = A \oplus B$	$\overline{C}$
$A \longrightarrow B$	$(\overline{A} + B)$	$A\overline{B}$	
$A \longrightarrow \overline{B}$	$(\overline{A} + \overline{B})$	AB	
$\overline{A} \longrightarrow B$	$(A + B)$	$\overline{A}\overline{B}$	
$\overline{A} \longrightarrow \overline{B}$	$(A + \overline{B})$	$\overline{A}B$	

All those invariants that have passed the base case will move into the induction step. In this step, we unroll the circuit into two time-frames which is illustrated in Fig. 3.2. The PPIs in the first window (assume window) are fully controllable. Then we insert the candidate invariant into the assume window and verify its negation in the second window (verify window). For example, consider Fig. 3.2, for invariant $A \rightarrow B$, a potential constraint $(A_0' + B_0)$ is assumed in the first window and the negation of the candidate, $(A_1)(B_1')$ is inserted in the second window. The subscripts for the variables indicate the time-frame. Table 3.4 is the conversion form. CNF column means the CNF representation for relation which should be inserted in assume window and objective column is the assertion we want to verify in second window. For equivalent relation $A = B$ or $A = \overline{B}$, we need to insert an additional XOR gate for assertion. This entire instance is given to the SAT solver. If the solver returns UNSAT, it means that the candidate invariant is true, since any state that holds the invariant cannot later enter a state that violates it.

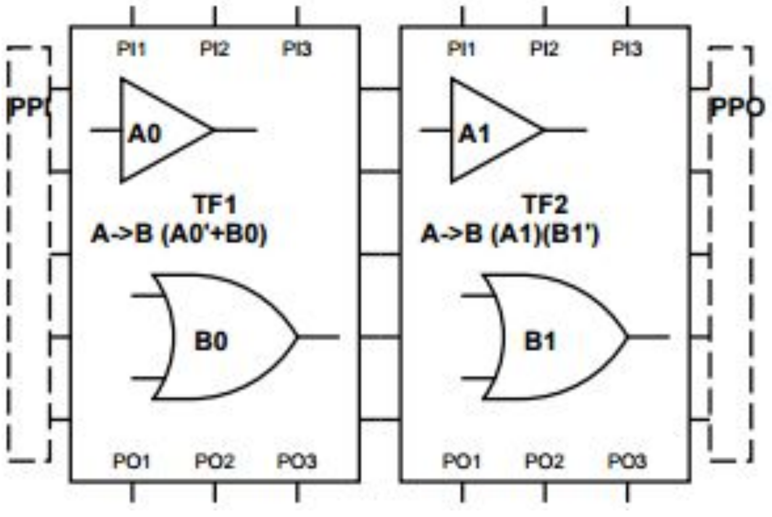


Fig.3.2 Assume-then-Verify Framework

On the other hand, if the instance is satisfiable, more information would be needed to determine if it is a false invariant. This is because the PPIs in the assume window are fully controllable, and any satisfying assignment returned by the SAT solver may include an unreachable state. Thus, such an assignment is actually an invalid witness with regard to what we are trying to prove. In order to reduce such spurious assignments, all potential invariants (that have passed the base case) are injected into assume window together. Then we perform the inductive step for all the candidate invariants one by one. If there is any violation in the verify step, we remove such relations from the assume window at the end of current iteration and repeat the entire procedure again using only remaining invariants. By repeating this process, a fixed point will eventually be reached in which a subset of the original potential invariants is proven as true inductive invariants. The computation time to reach the fixed-point will depend on the size of the potential invariants. If the set is huge, the proving time can be exceedingly large.

3.5 Efficient Filters

Given a set of potential invariants which needs to be verified, the computation time to reach self-sufficient set will be extremely expensive. Therefore, we could intelligently add different kinds of filters to avoid the SAT calling times and then accelerate the calculation process. Basically, there are two types of filters, static filter and dynamic filter. And we apply such filters both for 2-node and multi-node invariants. Note that some kinds of filters have been proposed in [32] and we also include them into our low lost filtering framework.

3.5.1 Static Filter

From previous chapter, we know that we obtain the potential invariant set by analyze the simulation database, thus there often exist lots of redundant implication invariant in the set which could not help us to further restrict the search space. For example, the input and output of a buffer will be a potential equivalent pair which however, will hold for the entire space. Without the information about the circuit structure, we may contain a huge number of such kind of relations and try to prove them in invariant checking process. Instead of that, we could obtain the implication graph first and add efficient static filter to delete such set in invariant candidate generation stage.

One kind of static filter is focusing on buffer and not gate type as mentioned before. For example in Fig. 3.3 (a), we have 5 potential implications:

$A = B\,(1) \quad A \longrightarrow \overline{C}\,(2) \quad B \longrightarrow \overline{C}\,(3) \quad A \longrightarrow \overline{E}\,(4) \quad B \longrightarrow \overline{E}\,(5)$

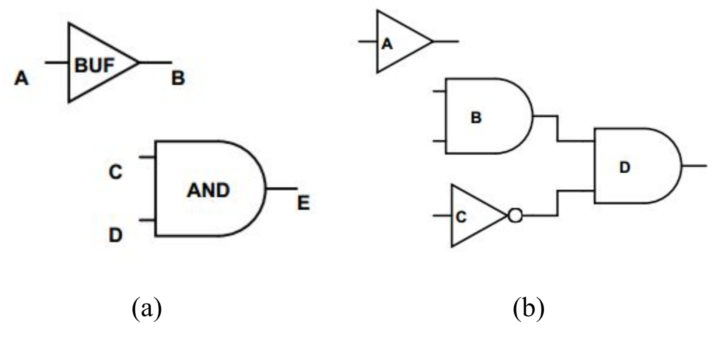


Fig. 3.3 Static Filter Example

Here, implication (1) is redundant since A and B are input and output of a buffer and it's true for all states. Invariant (2) and (3) are equivalent because from implication graph we will get $A \longrightarrow B$ and $B \longrightarrow A$, they could easily convert (2) and (3) from each other. Invariants (4) and (5) could be applied the same theory. Therefore, we could just keep 2 potential relations instead of 5. Generally, we just ignore all buffer and not gate in our invariant candidate generation process.

There is another kind of static filter which is shown in Fig. 3.3 (b). Assume we have the potential implication set:

$A \longrightarrow B$ (1) $A \longrightarrow \overline{C}$ (2) $A \longrightarrow D$ (3)

By analyzing structure relations among signal B, C and D, invariant (3) would be a true relation if relation (1) and (2) are both true. If either invariant (1) or (2) is false, relation (3) will be definitely wrong which means invariant (3) will not help shrink the search space formed by invariant (1) and (2) together any further. Therefore, we could filter invariant candidate (3) by static analysis. People may argue that we could keep invariant (3) only instead of (1) and (2) together. While in that case if invariant (3) is false, we will miss some implications since either invariant (1) or (2) still might be true. Such static filter could be used within different kinds of gates like AND output is logic 1, OR output is logic 0, NAND output is logic 0 and NOR output is logic 1.

When our framework goes into multi-nodes phase, the implication graph could also help us delete lots of true missing patterns. For example, if we get true implication $C \longrightarrow D$ in early stage which means pattern *CD* (10) is missing, any multi-node potential invariant contains such gate combination could be filtered such as *ABCD*(1110) and *CDE*(101 and 100). In our frame work, it could help reduce as much as 40% potential multi-node invariants.

3.5.2 Dynamic Filter

Different from static filter, dynamic filter is applied during SAT solver proving stage. In the invariant checking process, we need to check candidate by calling SAT solver for every potential relation if there is no dynamic filter. If any of them is false, we should delete it from our assume window at the end of current iteration and repeat the whole checking process based on all remaining invariants until we reach the fixed point. Obviously, it's very time consuming and make the checking process expensive. The motivation of dynamic filter here is to take advantage of both static implication graph and the invariant checker results we have already gotten from this iteration to reach a conclusion for any further candidate rather than calling SAT solver repeatedly. We have true invariant dynamic filter and false invariant dynamic filter here.

True Invariant Dynamic Filter

Let us explain this term using example shown in Fig. 3.4. Say we have 4 potential invariants here:

$A \longrightarrow B$ (1) $A \longrightarrow C$ (2) $A \longrightarrow \overline{D}$ (3) $A \longrightarrow E$ (4)

Assume invariant (1) is proven true in current iteration, without the circuit knowledge we will also need to call SAT solver to check invariant candidate (2), (3) and (4). However, from the static implication graph, we will get implication $B \longrightarrow C$ and due to property $A \longrightarrow B \longrightarrow C = A \longrightarrow C$, we could get the conclusion invariant (2) is true for such iteration without calling solver. We could apply similar method to candidate (3) and (4) as well. Therefore, we may totally call SAT solver 1 time instead of 4 times.

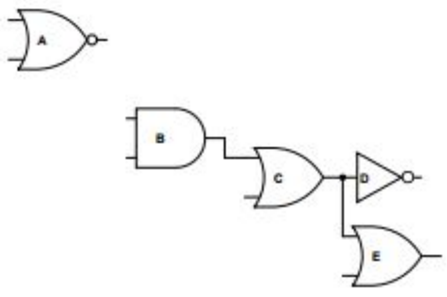


Fig. 3.4 Dynamic Filter Example

When the design is complex enough, we need to build static implication graph first. Then during every iteration for every candidate, we may face three conditions. (A) If there is a path in implication graph connecting the gates under proving, we could use dynamic filter to get the conclusion that the relation is true in current iteration. (B) Otherwise, we need to call SAT solver to execute the induction proving for this invariant, if the SAT solver returns UNSAT which means it's a true invariant for now, we add a new path into our implication graph. (C) Otherwise, if it's a false invariant, we also need to record the witness provided by SAT solver for further use. If it's not the fixed-point iteration, we should clear all induction invariant paths at the end of current iteration and repeat the checking process again.

The potential candidates order matters for true dynamic filter. For example in Fig. 3.4, if the candidate order is:

$A \longrightarrow E$ (1) $A \longrightarrow \overline{D}$ (2) $A \longrightarrow C$ (3) $A \longrightarrow B$ (4)

We need to call SAT solver 3 times since even though we prove invariant (1) is true and add path $A \longrightarrow E$ into implication graph, there is no path to prove other candidates. From our research, we tried to order candidates by several ways: natural order (from generation scheme directly), gate level order and cone size order and the result shows that nature order is the one to call dynamic filter most times and spend least calculation time for each iteration.

The true dynamic invariant filter is also applied in multi-node invariant checking phase. Basically, we have two approaches. One is taking advantage of true implication with fewer nodes. For example, assume we have proven pattern ABC(001) is missing, other missing patterns in our candidate set such as ABCE(0010) and ABCFGT(001000, 001110) could be filtered directly in the same iteration. The other kind of multi-node dynamic invariant filter is to use implication graph, say we have already concluded candidate EDF(000) is true, for candidate EDG(001), if we have a static implication path $\overline{F} \longrightarrow G$, we could also get a true conclusion without calling SAT solver.

Table 3.5 Dynamic Filter

Iteration ID	Up to 2-node potential invariant set	Up to 3-node potential invariant set
I1	7,647	15,459
I2	5,624	10,900
I3	5,298	9,667
I4	4,176 (fixed point)	8,542
I5		7,786
I6		6,377 (fixed point)

There is another kind of true dynamic filter which concerns multi-node potential invariant set from different phases. We will explain it using Table 3.5. For a design under verification, assume up to 2-node potential invariant set contains 20,982 relations and we will reach fixed-point after 4 iterations using assume-then-verify invariant checker. Column 2 in table shows the number of temporary true invariants for each iteration. If 2-node constraints are not sufficient to prove two designs are equivalent, we need to add 3-node potential invariant set as well, the total potential invariant number is 43,094. The third column in table is the number of temporary true invariant under each iteration. Since 3-node invariant checker will contain all up to 2-node implications, the temporary true invariants for 2-node frame-work should also be true in the same iteration for 3-node scheme without SAT solver calling. For example in second iteration, 5,624 true invariants out of 10,900 are filtered in this way. Generally, the filter could help us reduce 15% to 25% SAT solver calling times.

False Invariant Dynamic Filter

Next, we introduce a dynamic filter for false candidates proposed in [32]. When the SAT solver returns SAT for potential candidate, it will provide a witness. The same witness might falsify other candidates in potential implication set as well. Thus, we record the witness and use the same assignment from the witness to check other candidates, trying to reduce the number of future SAT solver calls, hence speed up the process. It could help us reduce false candidate early in every iteration with low cost.

3.6 Bounded Flip-flop Cone of Influence (BFCOI) and Multi-node Missing Pattern Invariants

Single- and two-node invariants that we introduced before may be not sufficient to constraint the search space enough to conclude sequential designs equivalence. Therefore, We shift to multi-node invariants phase. To balance the set size of potential multi-node invariants and proof efficiency, we just take advantage of missing pattern invariants on flip-flops which are in the cone of false invariants proved from early stage.

3.6.1 BFCOI

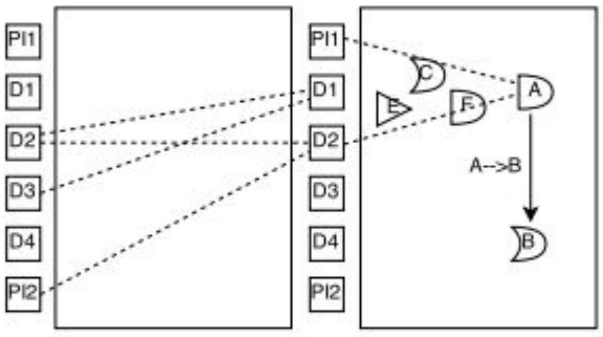


Fig. 3.5 BFCOI

Given a gate, *g*, in a circuit, its Cone of Influence (COI) is the set of all the gates that reside in *g*'s transitive fan-in cone, which may span multiple time-frames. The concept of COI is used to effectively reduce the complexity of a problem into a smaller, more manageable size.

Fig. 3.5 shows an example. Assume we want to calculate the COI for gate *A*. In single time-frame, $COI_{(A,\ 1tf)} = \{C, E, F, PI_1, D_1, D_2\}$ and the COI which just contains flip-flops is $FCOI_{(A,\ 1tf)} = \{D_1, D_2\}$. For flip-flop D_1 and D_2, however, their values are determined by related signals from previous time-frame, $FCOI_{(D1,\ 1tf)} = \{D_2, D_3\}$ and $FCOI_{(D1,\ 1tf)} = \{D_2, D_3, D_4\}$. Therefore, a union operation is performed for signal *A* involving two time-frames, $FCOI_{(A,\ 2tf)} = \{D_1, D_2, D_3, D_4\}$. The process is continued recursively until $FCOI_A$ reaches a fixed-point at which no more flip-flop will be added to this set. We call this set Complete FCOI (CFCOI). Note that for most designs, CFCOI may include a majority of flip-flops in the circuit. For the interest of this work, a Bounded FCOI (BFCOI) under two time-frames is applied. For the invariant $A \rightarrow B$ in Fig. 3.5, its BFCOI should be the union of $BFCOI_A$ and $BFCOI_B$. Let the $BFCOI_{(B,\ 2tf)}$ be $\{D_3, D_5\}$ then $BFCOI_{(A\text{-->}B,\ 2tf)} = \{D_1, D_2, D_3, D_4, D_5\}$. We also call such BFCOI a multi-node generation trace.

3.6.2 Multi-node Missing Pattern Invariants

With the BFCOIs computed, we start the generation of multi-node invariants from the potential 2-node invariant sets containing only flip-flops. After applying an invariant checker to the two-node invariant candidates, some are falsified. Consider a falsified relation $A \rightarrow B$ as an example. The fact that signal *A* seems to imply signal *B*, as observed in the database, indicates that it is difficult to obtain combination ($AB=10$) by simulation. However, the SAT solver could assign a witness that sets $AB=10$. The reason for this might be that the assignment on the BFCOI of $A \rightarrow B$ is not in the reachable state set. In other words, $A \rightarrow B$ might still be true if we could efficiently eliminate those false counter examples. The flip-flop set that is closely related to such an invariant has been computed before: $BFCOI_{(A\text{-->}B,\ 2tf)} = \{D_1, D_2, D_3, D_4, D_5\}$. We then generate multi-node invariants from this BFCOI.

[32] has proposed a multi-node invariants generation scheme among target flip-flop subsets which also takes advantage of false 2-node invariant traces. In their approach, however, such traces are used as a guide to construct the database. For instance, if a trace under verification containing 20 flip-flops, they may extract 3-, 4- or 5-node potential multi-node invariants among the 20 flip-flops.

It uses the same search algorithm as the previous multi-node generation framework on all circuit signals but focuses only on a subset of the flip-flop set.

However, our approach proposes a different generation algorithm that explores the same database to obtain a smaller multi-node invariant set. Using the running example *A→B*, our objective is to extract all missing patterns for D_1, D_2, D_3, D_4 and D_5. Searching through the database for 2^5=32 possible assignments ranging from 00000 to 11111 (0 to 31 in integer), suppose that 25 such patterns are found, leaving the other 7 missing (3, 5, 7, 8, 9, 13, 17). The missing patterns are important multi-node invariant candidates for three reasons: (1) there is a high probability that they are in the unreachable space which could prune reachable space further, (2) they have close relations to the target falsified invariant *A→B*, and (3) such FFs from the BFCOI will have a greater influence for the falsified invariant, compared to extracting arbitrary 3- or 4-node invariants from the flip-flips. Thus, if we pass those constraints together with the falsified 2-node invariant into the solver, they may help to prove the original invariant true.

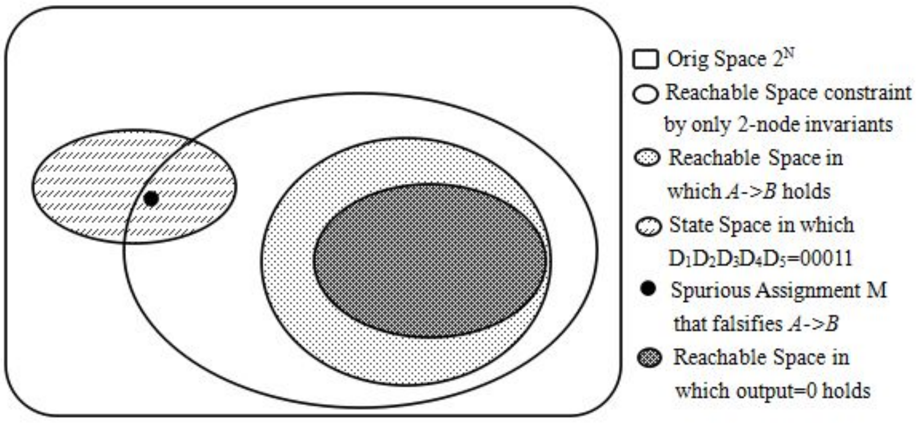

Orig Space 2^N
Reachable Space constraint by only 2-node invariants
Reachable Space in which *A->B* holds
State Space in which $D_1D_2D_3D_4D_5$=00011
Spurious Assignment M that falsifies *A->B*
Reachable Space in which output=0 holds

Fig.3.6 Shrink Reachable Space by Multi-node Invariants

Fig. 3.6 is an example to explain this concept. The universe defines the initial state space for the design which contains all 2^N states, where *N* is the number of flip-flops. After both single- and 2-node invariants have been learned and verified, we could prune the reachable space to the area specified by the largest ellipse using these true inductive invariants we have just obtained. However, the space is still much larger compared to the actual reachable space. We then use the falsified invariants, such as *A→B* to compute additional multi-node invariants among the FFs in the corresponding BFCOIs. Consider the previous example of BFCOI and multi-node invariant set {3, 5, 7, 8, 9, 13, 17}. Assume the solver returns a satisfiable assignment M in which

$D_1D_2D_3D_4D_5$=00011(3) is one of our missing pattern invariant candidates. When we combine this multi-node invariant together with invariant $A \rightarrow B$ as constraints into the solver, if the solver proves the missing pattern invariant is true, it could prune the search space to avoid the dashed ellipse area and therefore, efficiently eliminate the false counter example obtained before. By adding all invariants on the BFCOI we obtained, we will have a better chance to prune the reachable space into the region of the dotted ellipse, which will help to prove our target 2-node invariant $A \rightarrow B$ true.

Recall the simulation database being constructed in chapter 3.2, we could also use this technique to generate multi-node missing pattern invariants. For every trace in our record, we need to traversal the whole simulation database and find all the missing combinations. A two dimensional form in which column represents for FF ID and row represents for vector ID is shown in Table 3.6. Let's assume false invariant $P = 1$ involves 3 flip-flops. The objective is to extract all missing patterns for FF_1, FF_2 and FF_3 combination. The 8 possible assignments on these three flip-flops range from 000 to 111 (0 to 7 in integer). Searching through the database, we obtain 5 patterns (0, 1, 2, 4, 6) leaving the other 3 missing (3, 5, 7). The missing patterns become our potential multi-node invariants. If we simulate more input vectors, we will obtain more accurate result.

Table 3.6 Database for DFF

Vector ID	FF1	FF2	FF3
V1	0	0	1
V2	0	0	0
V3	1	0	0
V4	0	1	0
V5	1	1	0
V6	0	0	1
V7	0	0	0
V8	1	0	0
V9	1	0	0

3.6.3 Multi-node Invariants Checker

Similar to assume-then-verify invariants checker that we introduced in section 3.4, we also use bounded model checking with induction to prove multi-node missing pattern invariants. Again, we have base case checker and assume-and-verify checker. Since our candidates are only involved flip-flips in this section, we could ignore base case step.

In order to perform the assume-then-verify step, A 2 time-frames framework is needed as shown in Fig. 3.2. The PPIs here are in full scan mode which means in the first window (assume window), we could assign any state. Then we convert the relation into CNF and insert it into assume window and verify the objective in second window(verify window). For multi-node missing pattern invariant like *ACDF*(0001), we need to insert clause $(A_0 + C_0 + D_0 + \overline{F_0})$ into first window and assert 4 unit clauses $\overline{A_1} \cdot \overline{C_1} \cdot \overline{D_1} \cdot F_1$ in second window. The subscripts for the variables indicate the time-frame. Then the whole setup is passed to a SAT solver. If the SAT solver return UNSAT, such assignment is indeed a missing pattern and could help to shrink the space size. Otherwise, no conclusion could be drawn here. Since the design is in full scan mode (no constraint on flip-flops at the beginning), the multi-node missing pattern invariants may play a crucial role in constraining the initial state space and eliminating the false negative problem.

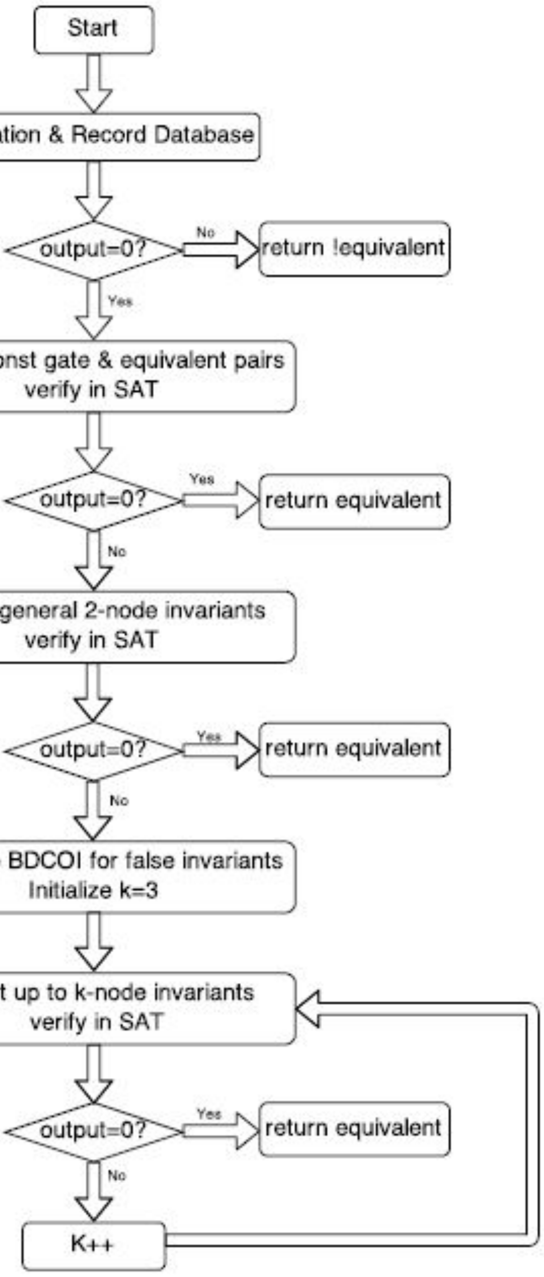
Again similar to 2-node invariant checker, we pass all false 2-node invariants as well as all multi-node invariant candidates into assume window in the first iteration. Whenever we get a witness from verify window, we need to delete such assumption and prove all the remaining candidates one more time until we reach the fixed-point.

3.7 Framework with Efficient Simulation-based Multi-Node Invariants

In this section, we will gather all the previous sections together and describe an efficient framework for our SEC engine which is shown in Fig. 3.7.

Our engine starts with sequential circuit random simulation. First of all, the circuit need to be simulated to an initial state, a state in which no flip-flop's value is *X*. Then we need to record all gate value information during random vector simulation as our database for further potential invariants generation.

The next step is to extract potential constant signal invariants and potential equivalent signal candidates. If the final output is not constant 0 during simulation, we could immediately conclude that the two designs are not equivalent. Otherwise, we will call assume-then-verify invariant checker to prove such candidates. After base case check, we send all potential candidates into assume window and check it one by one in the verify window until fixed-point. If output is constant 0 is in the final self-sufficient invariant set, we could say such true constant signal and equivalent signal invariants are sufficient to prove two designs are functionally equivalent. We are done. If not, we need more potential invariants and should shift to next phase.



Start
Simulation & Record Database
output=0?
No
return !equivalent
Yes
Extract const gate & equivalent pairs verify in SAT
output=0?
Yes
return equivalent
No
Extract general 2-node invariants verify in SAT
output=0?
Yes
return equivalent
No
Calculate BDCOI for false invariants Initialize k=3
Extract up to k-node invariants verify in SAT
output=0?
Yes
return equivalent
No
K++

Fig. 3.7 Our SEC Framework I

The following step is to calculate non-equivalent two-node relations just involving flip-lops. We then combine all the false candidates gotten from previous step and new potential invariants together as potential constraint set for this phase. After calling assume-then-verify invariant checker, if the true invariants are sufficient to prove two designs are equivalent, we could get our conclusion and exit the engine. Otherwise, we should go into multi-node invariant step.

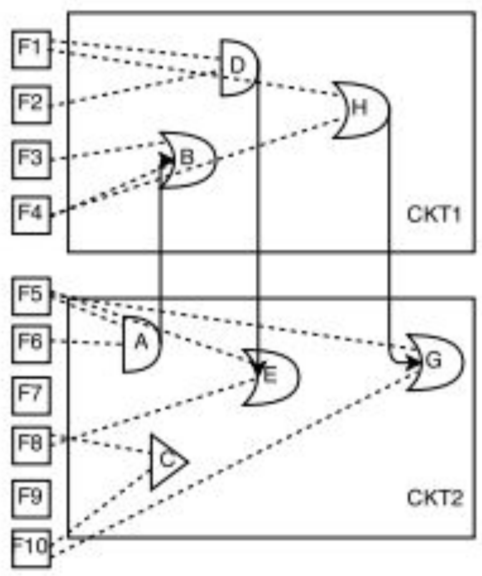


Fig. 3.8. Example of a Deep Invariant

In proving the multi-node invariants, we use a flexible threshold *K* to balance the size of the potential invariant set and proof efficiency. If the number of flip-flops in a BFCOI is less than or equal to *K*, we will search for all the missing pattern invariants among their flip-flops. Obviously, a larger *K* will provide us more potential invariants while we may also face the exponential problem. For example, a trace containing 4 flip-flops will generate at most 16 missing patterns, while up to 1,048,576 missing patterns will be generated when one trace contains 20 flip-flops. In addition, deeper invariants in the design may be partitioned into several shallow ones. A deep invariant refers to those signals are far away from flip-flops and thus a large number of flip-flops may be needed to prove it. For example, in Fig. 3.8, $H \rightarrow G$ is a deep invariant with its BFCOI size equal to 10. If we could successfully prove some shallow invariants within these 10 flip-flops by taking advantage of their BFCOIs, such shallow constraints may be sufficient to imply the deeper one as illustrated in the figure. In this figure, up to 6-node missing pattern invariants are generated and we could prove $A \rightarrow B$, constant *C* and $D \rightarrow E$ are true invariants. The subsequent reachable space then could be pruned by such invariants to a stage in which $H \rightarrow G$ also holds. In this example, invariants involving up to 6 FFs are needed to prove $H \rightarrow G$ instead of 10. Thus, we begin with a small *K* and increase it automatically when our potential multi-node invariant set is not sufficient. The initial *K* is set to 3. If

3-node invariants learned from the single- and 2-node falsified potential invariants are sufficient to prove the two circuits equivalent, we are done. Otherwise, we need to increase *K* by 1 and add all missing patterns gotten from 4-node BFCOI set. We apply this algorithm repeatedly until we have sufficient invariants to prove circuits equivalence or we exceed our limit (5 hours or 4GB).

3.8 Experimental Result

Our proposed SEC framework was implemented in C++ and the performance was evaluated on an Intel core I7-2700 3.5 GHz, 8GB RAM, running Ubuntu 12.04. Experiments are conducted for a suite of hard-to-verify SEC benchmarks generated using drastically different state encodings for ITC99 circuits. The state encodings chosen are gray-code encoding and one-hot encoding. Hence, we ensure that the two designs have very few or no structural similarities. Therefore, there are also few internal equivalent signals. In addition, combinations of different circuits are formed to make benchmarks T1 to T5 to verify our engine's efficiency for large scale design.

Table. 3.7 Results from Previous Approaches

Benchmark	#FFs	P1-2node		BLOSOM	
		# Potential Inv.	Time	# Potential Inv.	Time
B08	21/23	12,970/16,886	40	756	3.2
B09	28/30	22,246	8.5	189,702	2298
B10	17/24	17,290/15,014	30	801,246	31,841
B13	29/34	12,841/11,114	90.4	122,640	8,202
T1	66/66	70,790/72,826	6,219	N/A	N/A
T2	129/129	222,387/225,225	14,363/MO	N/A	N/A
T3	171/183	566,354	MO	N/A	N/A
T4	275/288	879,201	MO	N/A	N/A
T5	304/304	1,105,314	MO	N/A	N/A

P1-2node: 2-node Invariant Generation Algorithm proposed in [32]
BLOSOM: Multi-node Invariant Generation Algorithm proposed in [15]

Before we present our results, we will first provide a frame of reference of two recently published works that attempt to solve SEC problems using similar simulation based invariant generation schemes in Table 3.7. The first is a 2-node invariant generation algorithm among internal nodes both within and across time-frames [32], and the second is multi-node invariant generation scheme BLOSOM [15]. In this table, the first column lists the miter circuits, the second column reports the number of flip-flops under different encoding ways (gray and one-hot, namely). For example, 17/24 indicates that 17 flip-flops are in the first design and 24 are in the second one. The results for two previous approaches are shown in the following columns. For each miter, the number of potential invariants is reported, followed by proving time if the method could prove SEC successfully. In their approaches, either two- or multi-node invariants are mined from a sea of relations. For P1-2node, the number of potential invariants is shown as same time-frame / across time-frame. For example, the method in [32] (under P1-2node columns) first generates more than 222,000 2-node invariants from same time-frame for the T2 miter, taking 14,363s in total execution time. The true invariant set obtained was not sufficient for SEC verification and therefore, an extra 225,000 crossing-time-frame invariants are extracted. However, they were not able to prove SEC due to memory out. Similarity, more than 800,000 potential multi-node invariants are mined in BLOSOM for the B10 miter and it took more than 31,000 seconds to prove. These results indicate that an enormous amount of relations exists and significant computation cost is needed and they may or may not lead to conclusion.

Our results are shown in Table 3.8. The first column is the name of our miter circuit, the second column reports the flip-flop numbers under different encoding ways (similar to table 3.7). The third to sixth (under P1) are the result gotten using algorithm in[32]. The next 4 columns (under P2) are our result. For each method, we record 2-node invariant number, the minimal number *K* to get conclusion, multi-node invariant number and total run time. The time includes simulation, candidates extraction, assume-then-verify invariant checker calling, false invariant analysis, multi-node invariant checker and final SEC proving. The final column shows the speedup gotten by our algorithm as introduced before over P1.

Let's use T1 as an example. After random simulation, we could extract 1,390 potential up to 2-node invariants and we prove only 26 of them are true in current stage compared to 22 out of 4,400 using algorithm in [32]. Actually, this number should vary in a small range due to random input vectors, however, we could reduce a large percentage here since we do not contain potential const gate in the following extraction, which means if gate $A=1$ is potential const implications, we won't include $B \longrightarrow A$ or $\overline{C} \longrightarrow A$. For T1, such constraints are not sufficient to prove SEC, then we shift into multi-node invariant phase. In P1, up to 7-node invariants are needed which will add extra 7,439 implications and it takes 54s to prove equivalence. In our method, total 1,824 implications are added (up to 6-node) and we will spend only 12.7s to get the conclusion. 4.1 times speedup is achieved here.

We did not contain B01 to B04 and B11miter circuits in table because either they could be proven just by 2-node invariants or their flip-flop number is small so that we could easily find their true reachable state set. For B12, according to the best of our knowledge, there is no conclusion.

Obviously, we could reduce the simulation-based multi-node invariant number dramatically while still maintain a powerful constraint set to get the conclusion. Since invariant checker need to run several iterations to get fixed-point and traversal all remaining candidates in every iteration which is super time consuming, it's not surprised that our generation algorithm is more than 5× faster compared to previous mining-based methods. In fact, our technique works better on larger miter designs (T1 to T5) which makes it attractive.

Table 3.8 Our Experimental Result

Benchmark	#DFF	P1				P2				Speedup
		# 2-node Inv.	Up to k node	#Multi-node Inv.	Time	# 2-node Inv.	Up to k node	#Multi-node Inv.	Time	
B05	32/34	9,561/1,029	9	5,452	340	10,256/1788	4	4	131	2.6
B06	9/13	117/0	5	341	0.8	116/5	3	39	0.3	2.7
B07	51/53	17,872/281	14	81,733	9,613	19,121/142	N/A	N/A	N/A(MO)	N/A
B08	21/23	1,110/11	7	1,981	6	605/8	6	580	2.6	2.3
B09	28/30	2,169/78	6	800	6.1	2,487/92	6	546	5.1	1.2
B10	17/24	442/8	7	3,066	7.2	475/11	7	1654	3.6	2
B13	29/34	12,370/8,093	5	1,349	11.2	1,219/456	5	425	6.5	1.72
T1	66/66	4,409/22	7	7,439	52	1,390/26	6	1,824	12.7	4.1
T2	129/129	21,329/7,061	7	41,865	979	9,798/1,160	6	8,688	405	2.4
T3	171/183	32,503/15,984	6	4,580	306	18,993/2,352	6	1,949	166	1.8
T4	275/288	69,238/40,781	7	59,629	2,246	22,571/2,682	7	14,287	412	5.5
T5	304/304	104,246/53,304	7	103,113	14,156	39,213/3,218	7	39,888	4,340	3.3

P1: Algorithm proposed in [32]

P2: Our approach without the constrained logic synthesis to merge potential invariants

3.9 Conclusion

In this chapter, we propose a novel technique which reduces the potential simulation-based multi-node invariant set size while maintaining efficiency of the equivalence proof. Compared to previous methods, we treat different kinds of invariant candidates with customized generation algorithms. We first identify constant and equivalent signals among internal signals and flip-flops. Then, we extract general two-node non-equivalent relations from the flip-flop set. Thirdly, we utilize an intelligent algorithm to group flip-flops into smaller subsets for multi-node invariant generation in order to avoid an exponential growth of invariant candidates. Experimental results show the effectiveness of our approach on SEC under two functional equivalent circuits with completely different state-encodings and few internal equivalences. Our approach could be more than 5× faster compared to previous mining-based methods.

Chapter 4

Efficient Simulation-based Multi-node Invariants Generation and Pruning Framework II

In this chapter, we describe our simulation-based multi-node invariant set pruning framework. For hard SEC instances that have few structural similarities and different state encodings, we note that constant and equivalence invariants are insufficient to prove the circuits equivalent. Thus, general relations among signals are needed to provide additional constraints to prune the search space. However, there is an ocean of multi-node invariants, which will require an enormous computational cost to find and prove. To address these concerns, we propose a novel multi-node invariant generation scheme which balances the invariant candidate set size and proof efficiency by targeting on critical flip-flop subsets in last chapter. It is our interest to analyze the small while powerful multi-node invariant candidate set we got before and prune it even more. In this chapter, we reduce the number of multi-node potential invariants that need to be proved without losing pivotal constraints further by a constrained logic synthesis technique. Experimental results show that we could guarantee the same conclusion for instances compared to algorithm described in last chapter. Our approach is up to 20× faster compared to previous mining-based methods.

The rest of the chapter is organized as follows. Section 4.1 is our motivation. Section 4.2 introduces prime implicant calculation. Section 4.3 presents details for constraint logic synthesis on multi-node invariants. Section 4.4 describes the whole SEC framework that we propose. Section 4.5 presents experimental results, conclusion is followed in Section 4.6 and finally, we compared mining based SEC result with ABC.

4.1 Motivation

Given a set of potential multi-node invariants that needs to be verified, the computation time to reach fixed point depends on its size. For large sequential designs, we may have a huge number of potential invariant candidates which requires significant space and time resources. From last chapter, we have already obtained an potential multi-node invariant set with reasonable size that could prove SEC successfully. It is our interest here to explore if we could merges some multi-node invariants to reduce the set size even further while without missing critical ones. Due to the property of miter design in which there might exist a large unreachable space, the number of missing patterns under the same flip-flop subset should be large (50% to 90% of the full combination number as shown in Table 4.1), we could kindly merge some of potential missing pattern invariants together to reduce the SAT solver calling times and then speed up the verification process.

Table 4.1 Number of Missing Pattern Invariants
under the Same Flip-Flop Subset

BFCOI Combination	# Missing Patterns before Synthesis	Total Number of FFs Assignment
5 12 13 20	9	$2^4 = 16$
12 13 25 26	10	$2^4 = 16$
20 21 23 24 26	26	$2^5 = 32$
5 12 21 22 26	25	$2^5 = 32$

In the past, merging of such potential invariants might lose important relations if a merged relation turned out to be false. For example, consider the following three potential invariants. $D_1D_2D_3$ = 010 (1), $D_1D_2D_3D_4$ = 0101 (2) and $D_1D_2D_3D_4$ = 0100 (3). In traditional multi-node generation algorithm, we will just keep invariant (1) since it covers invariants (2) and (3). If this invariant $D_1D_2D_3$ = 010 is proved to be true, then invariants (2) and (3) are actually redundant and the merging is fine. However, if the solver falsifies invariant (1), one of invariants (2) and (3) might still be true. Thus, if we had merged them and represented them with just invariant (1), we would have missed either (2) or (3) as an important constraint.

Recall the generation process that we proposed in the previous chapter where we used a flexible threshold *K* to generate multi-node invariant candidates. If the number of flip-flops in a BFCOI is less than or equal to *K*, we will search for all the potential multi-node invariants. Otherwise, we skip the BFCOI temporarily. Assume that we need up to 6-node invariants for a given SEC instance. This means that for a smaller *K*, such as 4, no conclusion could be made. In this case, we'd like to know if we could carefully include some other invariants among larger-sized BFCOIs at a smaller *K* stage.

For example, if we include some extra prime implicants obtained from 5- or 6-node invariants when *K* is set to 4, we may include sufficient constraints for SEC and thus get conclusion much faster.

4.2 Prime Implicant Calculation

In Boolean logic, according to [37], a term *P* is an implicant of Boolean function *F* if *P* implies *F*. More precisely, *P* implies *F* if *F* takes the value 1 whenever *P* equals 1. For example, the function $f(x, y, z) = xy + x + z$ is implied by implicants *xy, x* and *z*. A prime implicant of a function is an implicant that cannot be covered by another implicant. Consider the same running example, *x* and *z* are two prime implicants for *f* while *xy* is not since it is covered by *x*.

```
void calculate_PI{
   vector <vector <int> > implicant;
   bool flag;
   do {
      flag=0;
      for(int i=0; i<minterm.size(); i++) {
         for(int j=i+1; j<minterm.size(); j++){
            if(diffbit(implicant[i], implicant[j])==1){
               /*some implicants could be merged*/
               flag=1;
               tag implicant[i], implicant[j];
            }
         }
      }
      update implicant using tag inforamtion;
   } while (flag=1);
}

int diffbit (vector<int> m1, vector<int> m2)
   {return the number of different bits;}
```

Fig. 4.1 Prime Implicant Calculation

There are several different algorithms to calculate prime implicant for a given function. K-map [38] is a graphic method to deal with small number of variables. In this section, we apply a constrained version of logic synthesis, shown in Fig. 4.1, to compute prime implicants among the potential multi-node invariants obtained under the same BFCOI. The basic idea is to traverse the entire implicant database and compare every pair in that set. If the two invariants have a Hamming distance of 1, we tag and merge them. For example, a BFCOI containing flip-flops 5, 12, 13, 20 will have 2^4=16 possible assignments ranging from 0000 to 1111 (0 to 15 in integer). Suppose 9 of them are found to be potential invariants S= {0, 5, 7, 8, 9, 10, 11, 14, 15}. We then traverse S to examine every pair in it. For example, 0 (0000) and 8 (1000) differ only in the first bit, we merge them into x000 at the end of current iteration. We execute the process repeatedly until there are no more implicants in our set that could be merged. For the running example, 2 iterations are needed to reach a fixed point, S_1 = {x000, 01x1, x111, 100x, 10x0, 10x1, 101x, 1x10, 1x11, 111x}, S_2 = {x000, 01x1, x111, 10xx, 1x1x}. We call the fixed point set S_2 prime implicant set.

4.3 Constraint Logic Synthesis on Multi-node Invariants

To prune the multi-node invariant candidate set further without losing important constraints, we first pick a new threshold *N* (should be larger than *K*). If the number of flip-flops in a BFCOI is less than or equal to *K*, we apply the multi-node generation algorithm described in last section; otherwise, if the number of flip-flops in a BFCOI is larger than *K* but smaller than or equal to *N*, we will process the generation algorithm to find the missing patterns and then gather all multi-node candidates into a constraint logic synthesis set. Due to the observation that the missing patterns within the same BFCOI in a miter circuit is large (50% to 80% as shown in Table 4.2), we could merge some missing patterns to reduce the candidate set size dramatically. The detailed algorithm is following.

First, we run logic synthesis on all the potential invariants under the same BFCOI to obtain the prime implicants that cover all of them as shown in Fig. 4.1. This step could often prune the multi-node candidate set size by more than 50% as Table 4.2 shows. Next, we examine all the derived prime implicants from different BFCOIs and delete any duplicate ones. By doing so, 10% to 20% more invariants could be discarded. For example in Table 4.2, the BFCOI containing 5, 12, 13, 20 and the BFCOI containing 5, 12, 21, 22, 26 both produce prime implicant *g5g12 = 10*, we could easily merge them to reduce potential invariant number. Thirdly, we add all remaining prime implicants whose number of signals is less than or equal to *K* into our original multi-node invariant candidate set as extra up to k-node invariants to help prune the search space further. For the same running example which includes 5 prime implicants after synthesis: *A* (x000), *B* (01x1), *C* (x111), *D* (10xx) and *E* (1x1x), we will include only *D* (10xx) and *E* (1x1x) if *K=2* or all five of them if *K=3*. If all the potential multi-node invariants we use are sufficient to prove SEC, proving is completed.

Otherwise, we repeat the whole process for a larger *K* and *N*. By doing so, we won't miss important multi-node invariants since we could re-generate multi-node invariants for target BFCOIs and add original invariants instead of including only the prime implicants in the set. In other words, if a potential invariant containing 4 variables is a critical relation for this SEC and prime implicant invariant that covers the potential invariant is false when *K* is 3, we will include such an invariant in its entirety when we increment *K* to 4.

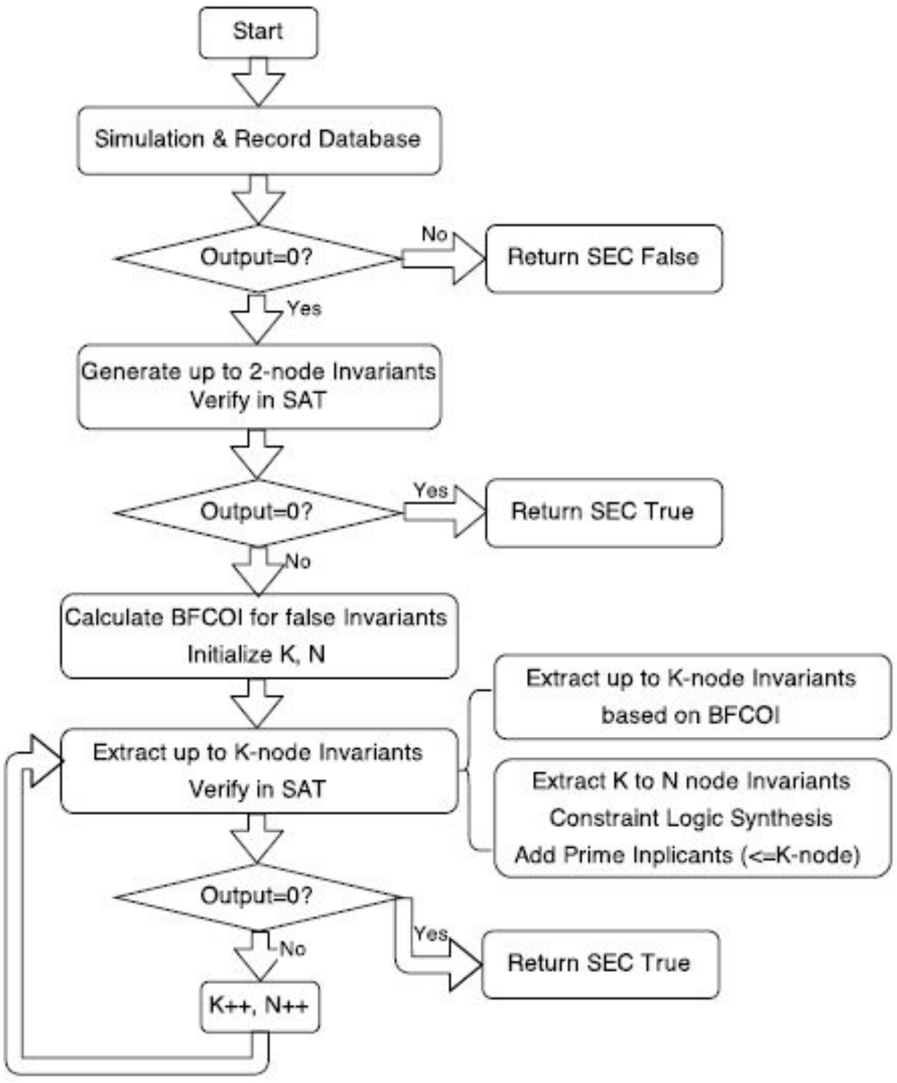
Table 4.2. Reduction in the Number of Potential Invariants
by using Constrained Logic Synthesis

BFCOI Combination	# Missing Patterns before Synthesis	# Missing Patterns after Synthesis
5 12 13 20	9	5
12 13 25 26	10	5
20 21 23 24 26	26	10
5 12 21 22 26	25	9

4.4 Our SEC framework

The complete framework for our SEC engine is shown in Fig. 4.2. Our engine starts with random simulation. From this simulation, after the circuit has reached an initial completely-specified state, we begin to record all gate value information during random vector simulation as our database for potential invariants generation. If the final output is not constant 0 at any step during the simulation, we could immediately conclude that the two designs are not equivalent. Otherwise, we extract single- and 2-node invariant candidates from this database. After invariant checker verification for the single- and two-node invariants, we apply SEC using only these proved invariants. If successful, such invariants are sufficient to prove this SEC instance. Otherwise, we need more constraints and should shift to the multi-node invariant checking phase.

Here, we use flexible threshold *K* and *N* to balance the potential invariant set size and proof efficiency as explained before. The initial *K* is set to 3. If all false candidates left from previous stage plus 3-node missing pattern invariants are enough to prove SEC, we are done. Otherwise, we could increase *K* and *N* automatically to include more invariants. The trade-off here is that if we need a large *K* to prove equivalence, we have to spend time calculating fixed-point for smaller *K*. We then add a new checkpoint to reduce unnecessary computational cost: at any point of our multi-node invariant checker verification process, if the SAT solver could set the miter output to 1, we could break from this iteration and increase *K* since the current value of *K* is insufficient to prove equivalence. The loop illustrated in Fig. 4.2 terminates when we either reach the conclusion or exceed our limit (time is longer than 5 hours or memory is larger than 4GB). If the algorithm times out (or memories out), there is no conclusion for this miter under verification.



Start
Simulation & Record Database
Output=0?
No
Return SEC False
Yes
Generate up to 2-node Invariants
Verify in SAT
Output=0?
Yes
Return SEC True
No
Calculate BFCOI for false Invariants
Initialize K, N
Extract up to K-node Invariants
based on BFCOI
Extract up to K-node Invariants
Verify in SAT
Extract K to N node Invariants
Constraint Logic Synthesis
Add Prime Inplicants (<=K-node)
Output=0?
Yes
Return SEC True
No
K++, N++

Fig. 4.2 Our SEC Framework II

4.5 Experiential Results

The proposed SEC framework was implemented in C++ and the performance was evaluated on an Intel core I7-2700 3.5 GHz, 8GB RAM, running Ubuntu 12.04. Experiments are conducted for a suite of hard-to-verify SEC benchmarks[23] generated using drastically different state encodings for ITC99 circuits. The state encodings chosen are gray-code encoding and one-hot encoding. Hence, we ensure that the two designs have very few or no structural similarities. Consequently, there are also few internal equivalent signals, as will be discussed later in this section. In addition, combinations of different circuits are formed to make benchmarks T1 to T5 to verify our engine's efficiency for large scale design.

Our results are reported in Table 4.3. The first column lists the miter circuits, the second column reports the number of flip-flop under different encoding ways (gray and one-hot, namely). For example, 17/24 indicates that 17 flip-flops are in the first design and 24 are in the second one. The third column gives the potential and true equivalent signal pairs extracted from the miter. As we mentioned before, for hard SEC instances that have few structural similarities and different state encodings, such constraints are insufficient. The fourth to sixth (under P1) columns report the results obtained using the algorithm proposed in [32] which are minimal *K* for successful SEC proving, the number of potential multi-node invariants and the total run time respectively. The next 5 columns (under P2) report our results without the constrained logic synthesis to merge potential invariants. The last five columns (under P3) report the results under constraint logic synthesis. For both P2 and P3, we record the minimal *K* to reach an SEC conclusion, the number of multi-node invariants (including both potential and true invariant) and the total run time. The time reported includes simulation, candidates extraction, invariant checker calling, false invariant analysis and final SEC proving. We also calculate the speedup for P2 and P3 over P1.

Let's use T1 as an example. After random simulation, we could extract 507 potential equivalent signal pair invariants and proved only 10 of them to be true. For T1, these 10 constraints are not sufficient to prove SEC. We then take the falsified 2-node invariants and prepare for the next step. Thirdly, we shift into multi-node invariant phase. In the previous approach, P1, up to 7-node invariants are needed (a total of 11,848 potential invariants) and it takes 54s to prove equivalence. In our approach P2 without constrained logic synthesis, a total of 3,214 potential invariants are added (up to 6-node) and 2,191 true invariants are obtained, 12.7s are needed to reach the conclusion. Our intelligent extraction of potential invariants leads to a much smaller potential set (in T1, it is only 27% compared to P1). Because the execution time includes setting up the simulation database, candidate extraction and other preprocessing steps, the speedup is not the ideal 16. In this case, a

4.1× speedup is achieved. Finally, by using constraint logic synthesis (Technique P3), we could reduce the number of potential multi-node invariants even further. For T1, up to just 4-node invariants are sufficient to prove SEC and total of 1,658 (nearly half compared to P2 and 13% compared to P1) potential invariants are used which lead 709 true invariants. Here, we could prove SEC in just 7.4s and this is 7× speedup over P1. In miter T2, the set of potential invariants needed is much larger than T1, In this case, the set of invariant candidates for P3 is less than 20% of P1, allowing for over 20× speedup.

The reason that we only needed a smaller *K* is that we run logic synthesis on missing pattern invariants whose size of flip-flops is larger than *K* but smaller than or equal to *N* and some of the prime implicants obtained within such set are critical for SEC in an early stage. However, in circuits such as B09, we may face the situation that such critical prime implicant invariants are false and could not provide us sufficient constraints, therefore, we still need the same *K* as shown in P2. Extra invariants are concluded (3,381 in P3 compared to 3,033 in P2) and we spend a longer time to reach a conclusion. However, our technique is still sound, that is, we guarantee to reach the same conclusion using our pruning algorithm.

Our approach is up to 20× faster compared to previous mining-based methods on large circuits. In fact, our pruning technique works better on larger miter designs (T1 to T5) which makes it more attractive.

Table 4.3. Our Experimental Results

Benchmark	#FFs	#Equivalent Pairs	P1			P2					P3				
			K	#Potential Invariants	Time	K	#Potential Invariants	#True Invariants	Time	Speed up	K	#Potential Invariants	#True Invariants	Time	Speed up
B08	21/23	152/0	7	3,091	6	6	1,185	600	2.6	2.3	4	666	165	1.6	3.8
B09	28/30	899/9	6	2,969	6.1	6	3,033	749	5.1	1.2	6	3,381	757	12.5	0.5
B10	17/24	86/3	7	3,508	7.2	7	2,129	1,631	3.6	2	5	842	598	2.3	3.1
B13	29/34	281/71	5	13,719	11.2	5	1,644	1,096	6.5	1.7	4	1,307	928	4.4	2.5
T1	66/66	507/10	7	11,848	52	6	3,214	2,191	12.7	4.1	4	1,658	709	7.4	7
T2	129/129	1,063/151	7	63,194	979	6	18,486	10,464	405	2.4	4	10,559	2,649	47	20.8
T3	171/183	4,198/1,410	6	37,083	306	6	20,942	1,934	166	1.8	4	19,045	1,375	109	2.8
T4	275/288	2,234/1,319	7	128,867	2,246	7	36,858	9,375	412	5.5	6	27,276	4,679	315	7.1
T5	304/304	5,970/1,390	7	207,359	14,156	7	79,101	19,102	4,340	3.3	6	48,401	8,104	862	16.4

P1: Algorithm proposed in [32]
P2: Our approach without the constrained logic synthesis to merge potential invariants
P3: Our approach with the constrained logic synthesis to merge potential invariants

4.6 Comparison with ABC

ABC is a growing software system for synthesis and verification of binary sequential logic circuits appearing in synchronous hardware designs developed by UC Berkeley[58]. SEC is also a core part of ABC. Since 2010, ABC has efficiently integrated Property Directed Reachability (PDR) based technology into their equivalence checking system and proposed a number of changes to the algorithm to improve its performance, which will dramatically improve their SEC capability. We then want to explore PDR method, compare its result with ours and then discuss the possibility to combine mining based SEC and PDR based SEC together.

Aaron Bradley [47] pioneered a novel method in sequential verification field a few years ago and he named his implementation IC3. [48] choose to call it Property Directed Reachability (PDR) to connect it to Bradley's earlier work on property directed invariant generation and implemented it into ABC system.

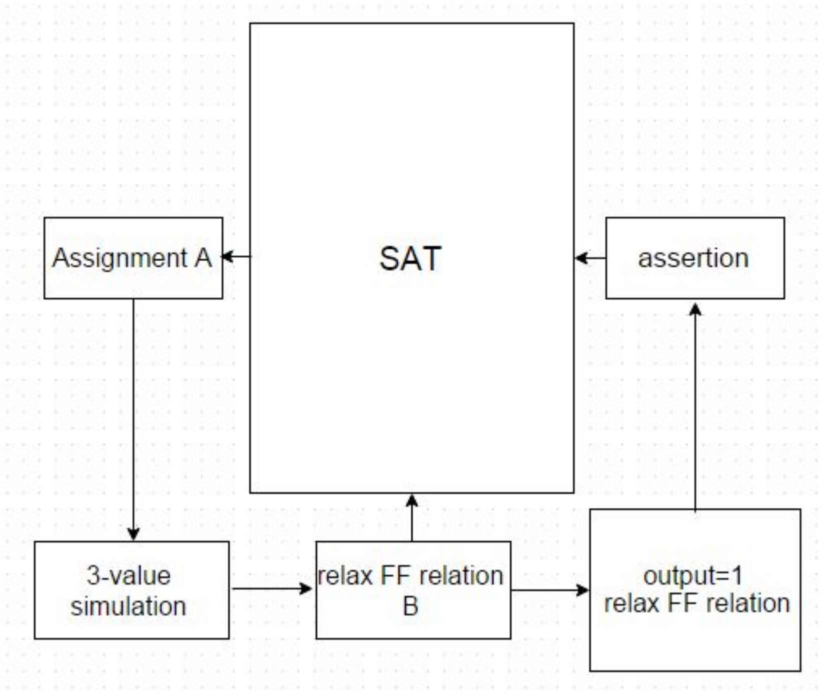
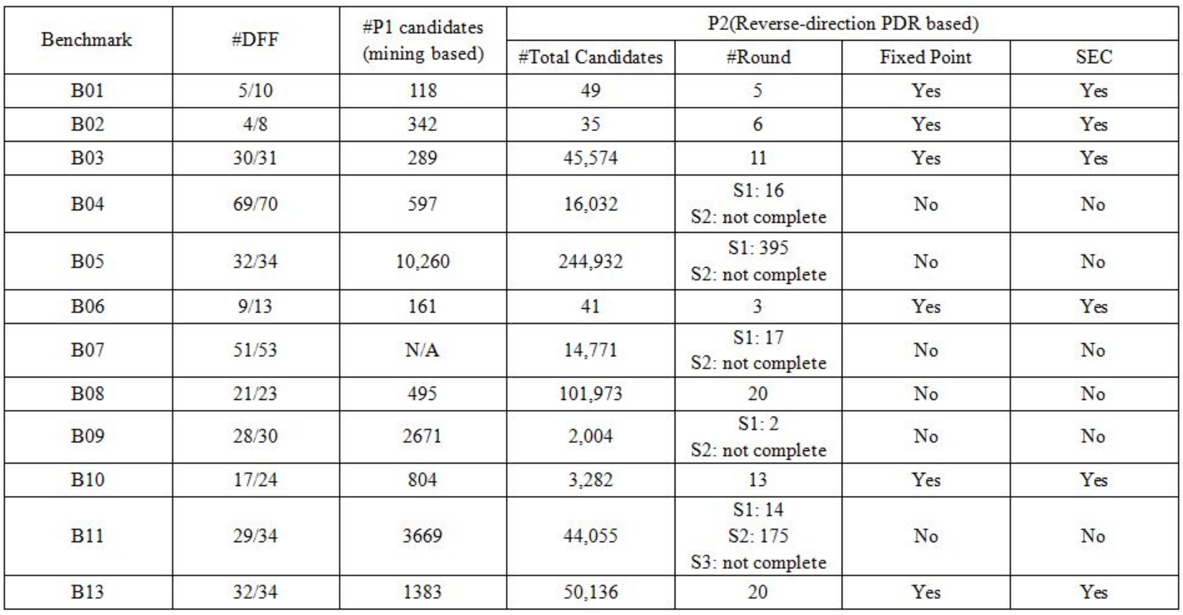


Fig. 4.3 Our Reverse-direction PDR framework

For SEC, the property is to prove the miter circuit output is constant 0. We follow a backward search path from the opposite phase of the property, generate states that could lead to such condition till fixed point, then if we could inductively prove output is constant 0 together with the property directed invariants, we could prove SEC successfully. Fig. 4.3 shows the detailed work flow. First, we set output=1 as our assertion and SAT will return us a counterexample A. Secondly, we apply assignment A into 3-value simulation engine to relax flip-flops, for any round, if relax a flip-flop value to X will lead output to 0 or X, we say it’s a key flip-flop for such assignment which could not be relaxed and record its value. We go through all the flip-flips with previous relaxing logic and get one relax FF relation B. State B could lead output=1 thus we call it property directed invariant. We then insert B’s corresponding clauses into SAT to prune search space that has already covered by it; at the meantime, we add B into our assertion set for further use. We follow the loop until SAT return UNSAT, which means there is no more assignment leads output to 1 in one step. All the relax FF relations now build our S1. Next, we could continue the process by picking up an assertion from S1 and keep generation relax FF relations, all the relations will then be stored in S2. We keep calculating until there is no new assignment, which means we reach the fixed point. Finally, if we could inductively prove output is constant 0 together with all the candidate constraints, we will get the conclusion the two sequential circuits are equivalent.

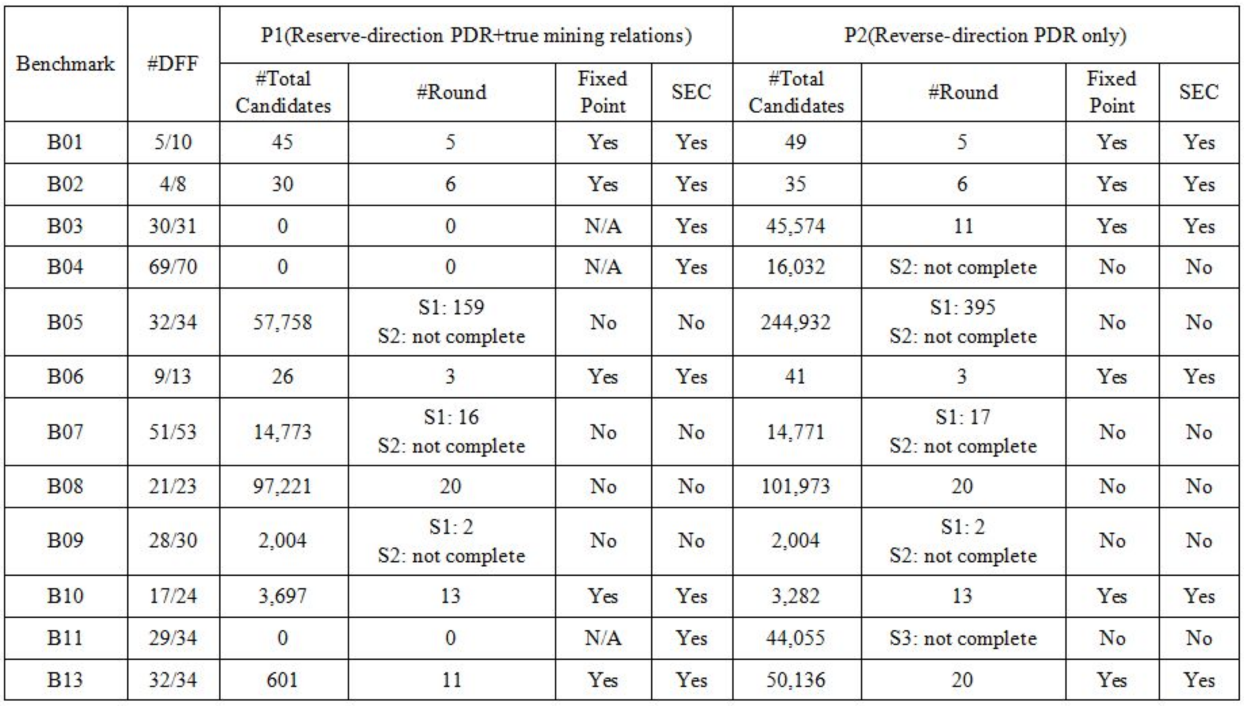
Table 4.4 Reverse-direction PDR result I



Benchmark	#DFF	#P1 candidates (mining based)	P2(Reverse-direction PDR based)			
			#Total Candidates	#Round	Fixed Point	SEC
B01	5/10	118	49	5	Yes	Yes
B02	4/8	342	35	6	Yes	Yes
B03	30/31	289	45,574	11	Yes	Yes
B04	69/70	597	16,032	S1: 16 S2: not complete	No	No
B05	32/34	10,260	244,932	S1: 395 S2: not complete	No	No
B06	9/13	161	41	3	Yes	Yes
B07	51/53	N/A	14,771	S1: 17 S2: not complete	No	No
B08	21/23	495	101,973	20	No	No
B09	28/30	2671	2,004	S1: 2 S2: not complete	No	No
B10	17/24	804	3,282	13	Yes	Yes
B11	29/34	3669	44,055	S1: 14 S2: 175 S3: not complete	No	No
B13	32/34	1383	50,136	20	Yes	Yes

Our results are reported in Table 4.4. We set a threshold to 1000 which means for a specific assertion, if we could generate more than 1000 relax FF relations, we will stop calculating such assertion and move on to next one. This will accelerate our PDR process while at the same time, we may not get to fixed point. In the table, the first column lists the miter circuits, the second column reports the number of flip-flop under different encoding ways (gray and one-hot, namely). The third column gives the mining based potential constraint number extracted from the miter. The fourth to seventh (under P2) columns report the results obtained using our reverse-direction PDR. For small circuit like B02, PDR will reach fixed point by just generating 35 relax FF relations and could prove SEC successfully. Compared to our mining based algorithm which need generate 342 multi-node constraint candidates, PDR is more efficient. However, for larger circuits, it's hard to prune output related flip-flops and relax them too much, which leads a huge relax FF invariant set to reach fixed point. For example in B05, we could generate 395 relax FF relations in S1, while most of relations in S1 will have more than 1,000 relax FF relations and when we get totally 244,932 constrains, it's still not fixed point and could not prove SEC successfully.

Table 4.5 Reverse-direction PDR result II



Benchmark	#DFF	P1(Reserve-direction PDR+true mining relations)				P2(Reverse-direction PDR only)			
		#Total Candidates	#Round	Fixed Point	SEC	#Total Candidates	#Round	Fixed Point	SEC
B01	5/10	45	5	Yes	Yes	49	5	Yes	Yes
B02	4/8	30	6	Yes	Yes	35	6	Yes	Yes
B03	30/31	0	0	N/A	Yes	45,574	11	Yes	Yes
B04	69/70	0	0	N/A	Yes	16,032	S2: not complete	No	No
B05	32/34	57,758	S1: 159 S2: not complete	No	No	244,932	S1: 395 S2: not complete	No	No
B06	9/13	26	3	Yes	Yes	41	3	Yes	Yes
B07	51/53	14,773	S1: 16 S2: not complete	No	No	14,771	S1: 17 S2: not complete	No	No
B08	21/23	97,221	20	No	No	101,973	20	No	No
B09	28/30	2,004	S1: 2 S2: not complete	No	No	2,004	S1: 2 S2: not complete	No	No
B10	17/24	3,697	13	Yes	Yes	3,282	13	Yes	Yes
B11	29/34	0	0	N/A	Yes	44,055	S3: not complete	No	No
B13	32/34	601	11	Yes	Yes	50,136	20	Yes	Yes

We then consider to prune our search space by adding static implication relation, true const gate constraints and true equivalent pair relations into SAT and we get result table 4.5. The first column lists the miter circuits, the second column reports the number of flip-flop under different encoding ways. The third to sixth columns (under P1) demonstrate the PDR plus extra constraints result. The seventh to tenth (under P2) columns report the results obtained using original PDR. Let's take B13 as an example, without prune search space, we need 50,135 relax FF relation to reach fixed point and then prove SEC; by constraint our search space, we will reduce the multi-node relation number to only 601. However, for some larger circuits, it's still hard to calculate a sufficient while efficient reverse-direction PDR relation set.

Table 4.6 ABC results

Benchmark	ABC (2007 version)		ABC (2011 version)					
	#DFF	SEC proven	#DFF	Interpolation-based SEC	BDD-based SEC	#PDR invariants	PDR-based SEC	Time
B01	15/0	Yes	15/0	Yes				0.12s
B02	12/14	No	12/11	No	Yes			119.85s
B03	60/0	Yes	60/0	Yes				0.06s
B04	139/0	Yes	139/0	Yes				0.13s
B05	66/103	No	66/53	No	Yes			97.30s
B06	22/18	No	22/11	Yes				0.26s
B07	106/0	Yes	106/0	Yes				0.93s
B08	44/0	Yes	44/0	Yes				0.15s
B09	58/88	No	58/58	No	Yes			244.34s
B10	41/72	No	41/40	No	Yes			19.42s
B11	75/0	Yes	75/0	Yes				0.46s
B13	63/55	No	63/24	Yes				3.63s
T1	132/35	No	132/21	Yes				0.89s
T2	258/143	No	258/89	No	No	179	Yes	19.62s
T3	354/105	No	354/69	No	Yes			42.48s
T4	563/176	No	563/109	No	No	208	Yes	17.14s
T5	626/230	No	626/130	No	No	247	Yes	16.32s

ABC performs better when combining its original sequential synthesis and verification algorithm together with PDR. It could solve most of our test cases in reasonable time. Table 4.6 provides the result. In this table, the first column lists the miter circuits. The following two columns indicates result from an old version ABC. The second column reports the number of flip-flops before and after re-synthesis and third column shows the SEC result. For example, 626/230 indicates that 626 flip-flops are in the original design and 230 are in the second one for T5 and ABC fail to conclude two circuits are equivalent. Note that several circuits will have 0 flip-flops after re-synthesis which means we could apply CEC to prove them easily. The following columns are the results gotten from a new version of ABC. In this framework, we list DFF number before and after re-structure, interpolation-based algorithm result, BDD-based method result, PDR-based invariant number, its conclusion and execution time which are from the forth to ninth columns. For example in T5, we could get 130 out of 626 flip-flops after re-structure, both interpolation-based or BDD-based algorithm fail to prove SEC, and then we go to PDR-based approach. It needs 247 invariants and proves SEC successfully in 16.32s.

We did some research to get a deeper understanding why ABC performs better. First, its PDR starts from a known initial state, and works forward by computing the approximate images until fixed point. It used the UNSAT-core to help relax the image, so that the fixed point can be reached faster than our backward approach. Second, ABC will re-synthesis the circuit before calling PDR. By doing so, it could reduce the flip-flop number dramatically or even convert into a CEC problem. In

B03, after re-synthesis, there is no flip-flop and prove such CEC only need 0.06s. In T5, the flip-flop number change form 626 to 130, which will naturally shrink the total search space from 2^{626} to 2^{130}. We also believe their other algorithms like BMC, interpolation, BDD will also help to narrow down the search space. Like Table 4.6 shows, most small circuits could be proven SEC without PDR.

4.7 Conclusion

In this section, we propose a novel simulation-based multi-node invariant generation and pruning method that balances the invariant candidate set size and proof efficiency. Compared to previous methods, we prune the potential invariant set without losing important constraints by cleverly merging some multi-node invariants with a constrained synthesis technique. In the past, merging of such potential invariants might inadvertently lose important relations if the merged relation turned out to be false. Our method mitigates this problem. Experimental results show the effectiveness of our approach on SEC under two functional equivalent circuits with completely different state-encodings and few internal equivalences. Our approach is up to 20× faster compared to previous mining-based methods for larger designs.

Chapter 5

Conclusion and Future Work

Verification is an important step for Integrated Circuit (IC) design. SEC, as one of the core verification tasks, plays a critical role in the design process since the designers might want to try different state encodings and optimization strategies and powerful equivalence checking could provide opportunities for more aggressive logic optimizations, meeting different goals such as smaller area, better performance, etc. A lot of research has been done in this field which made much progress in SEC, however, it still faces much challenge, especially for those complex circuits that have different state encodings and few internal signal equivalences.

In this thesis, we propose a novel simulation-based multi-node invariant generation and pruning method that balances the invariant candidate set size and proof efficiency. Compared to previous mining based methods, we treat different kinds of candidates with customized generation algorithms. For general multi-node invariants, in order to avoid an exponential growth of invariant candidates, we utilize an intelligent algorithm to group flip-flops into smaller subsets for invariant candidate generation. We then prune the potential invariant set without losing important constraints by cleverly merging some multi-node invariants with a constrained synthesis technique. In the past, merging of such potential invariants might inadvertently lose important relations if the merged relation turned out to be false. Our method mitigates this problem. Experimental results show the effectiveness of our approach on SEC under two functional equivalent circuits with completely different state-encodings and few internal equivalences. Our approach is up to 20× faster compared to previous mining-based methods for larger designs.

As for the future work, we would like to integrate our simulation-based invariant generation and pruning algorithm into a commercial SAT solver which is highly tuned for industrial designs. We believe with our methodologies, we can handle many SEC instances that are hard-to-verify. One way we can improve our framework is to apply re-structure technology before our SEC algorithm. For example in ABC, after re-synthesis, it could just leave 130 flip-flops out of 626 for T5. So if we could reduce flip-flop number by re-synthesis following the similar logic, it should naturally narrow down our search space at the beginning or even convert the problem into CEC.

Besides, it is also our interest to explore the combining algorithm between mining based constrains and property directed based invariants. We know there is still room to reduce the potential invariant set size while maintaining SEC proof efficiency, trying from both directions may provide us more insight. We did some attempts at the end of the forth chapter which demonstrates backward property directed method performs better in some conditions. We then prune the search space using some true invariants getting from our simulation algorithm together with reverse-direction PDR, it shows that some test cases could be handle better. We may get deeper into this field and bring some more intelligent ideas. We also know that ABC applies a forward direction PDR, from an initial state, it uses the UNSAT core to relax the image states computed until fixed point and the set size is small and target related. We may then want get an better understanding of difference between forward and backward PDR and also apply such forward direction PDR together with our mining based algorithm later to compare the result.

References

[1] S. Vijayaraghavan and M. Ramanathan. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2006.

[2] A. Sanghavi, "What is formal verification?". EE Times_Asia, May 2010

[3] J. Harrison, "Formal verification at Intel." In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pp. 45-54. IEEE, 2003.

[4] C. Baier and J.P. Katoen. *Principles of model checking*. Vol. 26202649. Cambridge: MIT press, 2008.

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation."*Computers, IEEE Transactions on* 100, no. 8 (1986): 677-691.

[6] S. Reda and A. Salem. "Combinational equivalence checking using Boolean satisfiability and binary decision diagrams." In *Proceedings of the conference on Design, automation and test in Europe*, pp. 122-126. IEEE Press, 2001.

[7] E. Goldberg, M. Prasad, and R. Brayton. "Using SAT for combinational equivalence checking." In *Proceedings of the conference on Design, automation and test in Europe*, pp. 114-121. IEEE Press, 2001.

[8] R. Arora and M. S. Hsiao. "Enhancing sat-based equivalence checking with static logic implications." In *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International*, pp. 63-68. IEEE, 2003.

[9] R. P. Lajaunie and M. S. Hsiao. "An effective and efficient ATPG-based combinational equivalence checker." In *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pp. 248-253. ACM, 2005.

[10] O. Coudert, "Verification of sequential machines using Boolean functional vectors." In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 179-196. 1989.

[11] C. Pixley, "A theory and implementation of sequential hardware equivalence." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 11, no. 12 (1992): 1469-1478.

[12] J. K. Zhao, E. M. Rudnick, and J. H. Patel. "Static logic implication with application to redundancy identification." In *VLSI Test Symposium, 1997., 15th IEEE*, pp. 288-293. IEEE, 1997.

[13] J. K. 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 *VLSI Design, 2001. Fourteenth International Conference on*, pp. 163-169. IEEE, 2001.

[14] R. Arora and M. S. Hsiao. "Enhancing sat-based bounded model checking using sequential logic implications." In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 784-787. IEEE, 2004.

[15] N. Goel, S. H. Michael, N. Ramakrishnan, and M. J. Zaki. "Mining complex Boolean expressions for sequential equivalence checking." In *Test Symposium (ATS), 2010 19th IEEE Asian*, pp. 442-447. IEEE, 2010.

[16] W. Kunz and D. K. Pradhan. "Accelerated dynamic learning for test pattern generation in combinational circuits." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 12, no. 5 (1993): 684-694.

[17] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 29. wh freeman, 2002.

[18] M. R. Prasad, A. Biere, and A. Gupta. "A survey of recent advances in SAT-based formal verification." *International Journal on Software Tools for Technology Transfer* 7, no. 2 (2005): 156-173.

[19] M. Davis, G. Logemann and D. Loveland. "A machine program for theorem-proving." *Communications of the ACM* 5, no. 7 (1962): 394-397.

[20] M. Davis and H. Putnam. "A computing procedure for quantification theory." *Journal of the ACM (JACM)* 7, no. 3 (1960): 201-215.

[21] J. P. M. Silv and K. A. Sakallah. "GRASP—a new search algorithm for satisfiability." In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pp. 220-227. IEEE Computer Society, 1997.

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an efficient SAT solver." In *Proceedings of the 38th annual Design Automation Conference*, pp. 530-535. ACM, 2001.

[23] N. Eén and N. Sörensson. "An extensible SAT-solver." In *Theory and applications of satisfiability testing*, pp. 502-518. Springer Berlin Heidelberg, 2004.

[24] N. Sörensson and N. Eén. "Minisat v1. 13-a sat solver with conflict-clause minimization." *SAT* 2005 (2005): 53.

[25] N. Eén and A. Biere. "Effective preprocessing in SAT through variable and clause elimination." In *Theory and Applications of Satisfiability Testing*, pp. 61-75. Springer Berlin Heidelberg, 2005.

[26] Max Roser (2014) – 'Technological Progress'. Published online at OurWorldInData.org. Retrieved from: http://ourworldindata.org/data/technology-and-infrastructure/moores-law-other-laws-of-exponential-technological-progress/

[27] http://techupdates.in/very-large-scale-integration-technology-and-its-design-flow/

[28] S. A. Cook. "The complexity of theorem-proving procedures." In *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151-158. ACM, 1971.

[29]R. M. Karp. *Reducibility among combinatorial problems*. springer US, 1972.

[30] F. Luand K-T. Cheng. "Sequential equivalence checking based on K-th invariants and circuit SAT solving." In *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, pp. 45-51. IEEE, 2005.

[31] C. L. Chang, C. P. Wen, and J. Bhadra. "Speeding up bounded sequential equivalence checking with cross-timeframe state-pair constraints from data learning." In *Test Conference, 2009. ITC 2009. International*, pp. 1-8. IEEE, 2009.

[32] H. Nguyen, and M. S. Hsiao. "Sequential equivalence checking of hard instances with targeted inductive invariants and efficient filtering strategies." In*IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 1-8. 2012.

[33] M. Hazewinkel, ed. (2001), "Mathematical induction", Encyclopedia of Mathematics,Springer, ISBN 978-1-55608-010-4

[34] M. Sheeran, S. Singh, and G.   Stålmarck, "Checking safety properties using induction and a SAT-solver." In *Formal Methods in Computer-Aided Design*, pp. 127-144. Springer Berlin Heidelberg, 2000.

[35] L. De Moura, H. Rueß, and M. Sorea. "Bounded model checking and induction: From refutation to verification." In *Computer Aided Verification*, pp. 14-26. Springer Berlin Heidelberg, 2003.

[36] G. Audemard, and L. Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers." In *IJCAI*, vol. 9, pp. 399-404. 2009.

[37] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[38] M. Karnaugh, "The map Method for Synthesis of Combinational Logic Circuit", *Transactions of the American Institute of Electrical Engineers, Communications and Electronics*, Vol. 72, pp. 593-599, November 1953

[39] Berman, C. Leonard, and L. H. Trevillyan. *Functional comparison of logic designs for VLSI circuits*. Springer US, 2003.

[40] D. Brand, "Verification of large synthesized designs." In *The Best of ICCAD*, pp. 65-72. Springer US, 2003.

[41] W. Kunz, "HANNIBAL: An efficient tool for logic verification based on recursive learning." In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pp. 538-543. IEEE, 1993.

[42] S. Y. Huang, K. T. Cheng, K. C. Chen, C. Y. Huang, and B. Forrest, "AQUILA: An equivalence checking system for large sequential designs." *Computers, IEEE Transactions on* 49, no. 5 (2000): 443-464.

[43] W. Wu, and M. S. Hsiao. "Mining global constraints for improving bounded sequential equivalence checking." In *Proceedings of the 43rd annual Design Automation Conference*, pp. 743-748. ACM, 2006.

[44] J. P. Roth, “Diagnosis of automata failures: A calculus and a method, ”*IBM Journal of Research and Development*, vol. 10, pp. 278-291, July 1966.

[45] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits,” *IEEE Trans. on Comp.*, vol. C-30, pp. 215-222, March 1981.

[46] Biere, Armin, and Wolfgang Kunz. "SAT and ATPG: Boolean engines for formal hardware verification." In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pp. 782-785. ACM, 2002.

[47] A. R. Bradley. "SAT-based model checking without unrolling." In *Verification, Model Checking, and Abstract Interpretation*, pp. 70-87. Springer Berlin Heidelberg, 2011.

[48] N. Een, A. Mishchenko, and R. Brayton. "Efficient implementation of property directed reachability." In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pp. 125-134. IEEE, 2011.

[49] M. Rice and S. Kulhari. "A survey of static variable ordering heuristics for efficient bdd/mdd construction." *University of California, Tech. Rep* (2008).

[50] B. Bollig, et al., Improving the Variable Ordering of OBDDs is NP-Complete. IEEE Transactions on Computers, 1996.

[51] M. Syal, K. Chandrasekar , V. Vimjam, M. S. Hsiao, Y.-S. Chang, and S. Chakravarty, "A study of implication based pseudo functional testing," in *Proceedings of the IEEE International Test Conf.*, paper 24.3, October 2006.

[52] X.Liu and M. S. Hsiao, "Constrained ATPG for broadside transition testing," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 2003, pp. 175-182.

[53] M. S. Hsiao, "Maximizing impossibilities for untestable fault identification," in *Proceedings of the IEEE Design Automation and Test in Europe Conference*, March, 2002, pp. 949-953.

[54] S. Prabhakar and M. S. Hsiao, "Multiplexed trace signal selection using non-trivial implication-based correlation," in *Proceedings of the IEEE Symposium on Quality Electronic Design*, March, 2010, pp. 697-704.

[55] S. Prabhakar and M. S. Hsiao, "Using non-trivial logic implications for trace buffer-based silicon debug," in *Proceedings of the IEEE Asian Test Symposium*, November 2009, pp. 131-136.

[56] L. Zhang, M. R. Prasad, and M. S. Hsiao, "Incremental deductive and inductive reasoning for SAT-based bounded model checking," in *Proceedings of the IEEE International Conference on Computer Aided Design*, November 2004, pp. 502-509.

[57] M. Banga and M. S. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *Proceedings of the IEEE Hardware-Oriented Security and Trust Symposium*, June 2010, pp. 56-59.

[58] http://www.eecs.berkeley.edu/~alanmi/abc/