ENHANCING SAT-BASED FORMAL VERIFICATION METHODS USING GLOBAL LEARNING

By

Rajat Arora

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 Sciences
In
Computer Engineering

Dr. Michael S. Hsiao, Chair
Dr. Dong S. Ha, Member
Dr. Sandeep K. Shukla, Member

10th May 2004
Bradley Department of Electrical and Computer Engineering,
Blacksburg, Virginia

Keywords:
Boolean Satisfiability (SAT), Static Logic Implications, Combinational Equivalence Checking (CEC), Bounded Model Checking (BMC), Propositional Formula.

ENHANCING SAT-BASED FORMAL VERIFICATION METHODS USING GLOBAL LEARNING

RAJAT ARORA

ABTRACT

With the advances in VLSI and System-On-Chip (SOC) technology, the complexity of hardware systems has increased manifold. Today, 70% of the design cost is spent in verifying these intricate systems. The two most widely used formal methods for design verification are Equivalence Checking and Model Checking. Equivalence Checking requires that the implementation circuit should be exactly equivalent to the specification circuit (golden model). In other words, for each possible input pattern, the implementation circuit should yield the same outputs as the specification circuit. Model checking, on the other hand, checks to see if the design holds certain properties, which in turn are indispensable for the proper functionality of the design. Complexities in both Equivalence Checking and Model Checking are exponential to the circuit size.

In this thesis, we firstly propose a novel technique to improve SAT-based Combinational Equivalence Checking (CEC) and Bounded Model Checking (BMC). The idea is to perform a low-cost preprocessing that will statically induce global signal relationships into the original CNF formula of the circuit under verification and hence reduce the complexity of the SAT instance. This efficient and effective preprocessing quickly builds up the implication graph for the circuit under verification, yielding a large set of logic implications composed of direct, indirect and extended backward implications. These two-node implications (spanning time-frame boundaries) are converted into two-literal clauses, and added to the original CNF database. The added clauses constrain the search space of the SAT-solver engine, and provide correlation among the different variables, which enhances the Boolean Constraint Propagation (BCP). Experimental results on large and difficult ISCAS'85, ISCAS'89 (full scan) and ITC'99 (full scan) CEC instances and ISCAS'89 BMC instances show that our approach is independent of the state-of-the-art SAT-solver used, and that the added clauses help to achieve more than an order of magnitude speedup over the conventional approach. Also, comparison with Hyper-Resolution [Bacchus 03] suggests that our technique is much more powerful, yielding non-trivial clauses that significantly simplify the SAT instance complexity.

Secondly, we propose a novel global learning technique that helps to identify highly non-trivial relationships among signals in the circuit netlist, thereby boosting the power of the existing implication engine. We call this new class of implications as 'extended forward implications', and show its effectiveness through additional untestable faults they help to identify.

Thirdly, we propose a suite of lemmas and theorems to formalize global learning. We show through implementation that these theorems help to significantly simplify a generic CNF formula (from Formal Verification, Artificial Intelligence etc.) by identifying the necessary assignments, equivalent signals, complementary signals and other non-trivial implication relationships among its variables. We further illustrate through experimental results that the CNF formula simplification obtained using our tool outshines the simplification obtained using other preprocessors.

To My Parents and Brother

- who have supported and guided me throughout .....

Acknowledgements

It is a pleasure to acknowledge all the people who made this work possible. I would like to express my sincere thanks to my advisor Dr. Michael S. Hsiao, for his inspiration and support throughout my graduate program. I would like to thank Dr. Dong S. Ha and Dr. Sandeep K. Shukla for serving on my thesis committee. Also, I thank my friends and relatives for their emotional support and encouragement throughout my stay at Virginia Tech. Last, but not the least, I would like thank my family members for their blessings, support and guidance. And finally, I would like to thank the Almighty for helping me come so far in life.

CONTENTS

CHAPTER 6

CHAPTER 7

LIST OF FIGURES

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

LIST OF TABLES

CHAPTER 6

CHAPTER 1

INTRODUCTION

With the advances in VLSI and System-On-Chip (SOC) technology, the complexity of digital systems has increased manifold. Verification of these intricate systems has become one of the foremost concerns for the validation and verification engineers. The two most widely used formal methods for design verification are Model Checking and Equivalence Checking. In the last few decades, Model Checking based verification [Clarke 86, Burch 90, McMillan 93, Boppana 99, Biere 99, Clarke 02] has gained much attention. It determines if the implemented design satisfies a given set of properties which in turn are indispensable for the proper functionality of the design. Binary Decision Diagram (BDD)-based Symbolic Model Checking [Burch 90, McMillan 93] has shown to hold promise. However, BDDs are known to suffer from the *memory explosion* problem, and hence fail for bigger circuits with large numbers of flip-flops/state variables. Automatic Test Pattern Generation (ATPG)-based Unbounded Model Checking [Boppana 99] on the other hand can suffer from *temporal explosion*. With the recent advances in Satisfiability (SAT) solvers [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97], SAT-based Bounded Model Checking (BMC) [Biere 99, Gupta 03, Cabodi 03] is gaining significant importance. In this technique, the sequential circuit is unrolled into k time-frames, and counterexamples (or bugs) are searched in this bounded length k. Comparisons of SAT-based approach with Sequential ATPG and BDD-based approaches can be found in [Saab 03] and [Cabodi02], respectively. In the last decade, a large number

of other problems in Electronic Design Automation (EDA) domain are also being modeled as Boolean Satisfiability (SAT) problems; Combinational Equivalence Checking (CEC) [Lu 03a, Novikov 03, Silva 99b, Silva 99c], Automatic Test Pattern Generation (ATPG) [Larabee 92, Lu 03b, Stephan 96] etc. being a few. The state-of-the-art SAT solvers [Moskewicz 01, Goldberg 02a, Ryan 03] are descendants of the DPLL-algorithm [Davis 62] and are usually based on the Conjunctive Normal Form (CNF). This form consists of the logical AND (conjunction) of one or more *clauses*, such that each *clause* is a logical OR (disjunction) of one or more *literals*. A *literal* may be a variable in its true or complement form. For the CNF formula to be *satisfied,* each of the individual clauses should be satisfied (*sat*). Each of these clauses are also called *implicates* of the CNF formula. While trying to satisfy this given CNF formula, the SAT-solver makes decisions based on a given set of variable selection heuristics [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97]. It learns dynamically from the conflicts encountered during the search and generates conflict-induced clauses [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97] that can subsequently constrain the search. However, the conflict clauses learnt dynamically have the following disadvantages:

1. Not all the learned clauses are useful, especially the long clauses.
2. The set of all learned clauses can grow very large.
3. The clauses are learned gradually over the entire SAT search, which may take a long time.

In this thesis, we work to overcome the above disadvantages to some extent by indulging in *static learning*. We do a quick preprocessing on the circuit netlist and deduce signal relationships (called static logic implications) which when introduced as binary

clauses into the existing CNF database of the circuit under verification, improve the performance of the SAT-solver. We term this static learning as *global learning*.

We also propose a way to identify non-trivial relations among signals in the circuit netlist, resulting in a new class of implications called *extended forward implications*. This new set of implications further enhances the power of existing implication engine, and can be used for various applications such as CEC, BMC, untestable fault identification, path delay testing, ATPG, logic synthesis etc.

Finally, we present a suite of Lemmas and Theorems to formalize *global learning.* These theorems help to significantly simplify a generic CNF formula by identifying the necessary assignments, equivalent signals, complementary signals and other non-trivial implication relationships among its variables. It is intuitive that when such a simplified CNF formula will be given to the SAT-solver for processing, the performance improvement will be considerable.

1.1 Previous Work

In recent years, efforts have been made to indulge in some sort of static preprocessing on the original CNF formula before the SAT solver starts. These efforts have enabled to overcome the drawbacks of dynamic learning to some extent. In [Novikov 03], the author introduced a technique that involved branching on small subsets of CNF variables, and analyzing the results of unit propagation. A restricted version of this technique was implemented, which focused on deducing only the constant values and equivalence relationships. In [Li 00], equivalence reasoning was integrated into the Davis-Putnam procedure [Davis 62] to enhance its performance on problems containing

equivalence clauses. In [Gupta 03], which focuses on improving SAT-based BMC, local BDDs were used to capture relationships among the Boolean variables of the CNF formula in the form of a characteristic function. The nodes/variables for which BDDs were created were termed as *seed nodes*, and these were selected statically or dynamically during the decision phase. Every path leading to the terminal node *0* in the resulting local BDD denoted a conflict, and the negation of the corresponding literals was added as a multi-literal learned clause to the existing CNF formula. However, the locally built BDDs were not helpful in extracting the global relations. In [Cabodi 03] which also tried to improve SAT-based BMC, the authors performed BDD-based approximate reachability analysis to gather information on the state space. This state space related information was converted to clauses and appended to the original CNF formula, which in turn restricted the search space of the SAT-solver. Probing-based preprocessing techniques for manipulating propositional satisfiability formulas was proposed in [Lynce 03]; meaningful information was inferred from a table of triggering assignments which was built by assigning a value to each of the variables and carrying out unit propagation. The technique also subsumed the additional binary clauses obtained in [Gelder 93].

More recently, in [Bacchus 02, Bacchus 03], preprocessing based on Hyper-Resolution and Equality Reduction was explored. The Hyper-Resolution technique takes as input the following:

(*a*) a single *n*-ary clause ($n \geq 2$), i.e. ($l_1 \vee l_2 \vee l_3 \ldots l_n$), and

(*b*) $n - 1$ binary clauses each of the form ($\neg l_i \vee l$) where ($i = 1, \ldots, n - 1$).

It then produces as output a new *binary clause* ($l \vee l_n$). For example, using Hyper-Resolution on the inputs ($a \vee b \vee c \vee d$), ($h \vee \neg a$), ($h \vee \neg c$), and ($h \vee \neg d$), the new binary

clause ($h \vee b$) is produced. Hyper-Resolution is equivalent to a sequence of ordinary resolution steps (i.e., resolution steps involving only two clauses). However, a sequence of ordinary resolution steps would generate clauses of intermediate length while Hyper-Resolution side-steps this to only generate a final binary clause. In a SAT-solver it is generally counter-productive to add these intermediate clauses to the CNF database, but it can be very useful to add the final binary clause. The above resolution steps also help to generate *unit clauses* (clauses with only one literal) which further simplify the CNF formula. Their preprocessing algorithm also performs equality reduction if the CNF database has equivalent literals. For example, if the CNF formula contains ($\neg a \vee b$) as well as ($a \vee \neg b$) (i.e., $a \Rightarrow b$ as well as $b \Rightarrow a$), then by equality reduction b can be replaced with a. The steps involved in equality reduction are:

- replacing all instances of b in CNF formula by a,
- removing all clauses which now contain both a and $\neg a$,
- removing all duplicate instances of a (or $\neg a$) from all clauses.

This process might generate new binary clauses. The hyper resolution technique was shown to be highly effective on a large variety of SAT benchmarks. However, the complexity of hyper resolution increases significantly as the number of clauses in the CNF formula increase. Hence, it is not viable for SAT-based BMC.

In [Silva 99b, Silva 99c], the Recursive Learning technique [Kunz 92, Kunz 93] was incorporated into SAT-solvers and applied to combinational equivalence checking (CEC). The Recursive Learning technique is guaranteed to find all possible necessary assignments in the circuit, given enough levels of recursion. However, as the depth of recursion increases, the time to compute the implications increases exponentially. As a

result, in [Silva 99b, Silva 99c], the authors preprocessed the CNF formula using only depth *one* Recursive Learning [Kunz 92, Kunz 93]. In [Lu 03a], which also focuses on improving SAT-based CEC, probable correlation among signal pairs was first obtained via random simulation on the miter circuit. Then, explicit learning was performed wherein the correlated signal pairs were assigned values that would most likely result in conflict. A SAT-solver was invoked to quickly learn a fixed number of conflict-induced clauses, corresponding to every pair of possibly correlated signals. Because random simulation was used, only a subset of signal correlations could be identified.

Schulz *et al.* were the first to improve the quality of implications by computing indirect implications in SOCRATES [Schulz 88]. In order to improve the quality of these indirect implications, static learning was extended to dynamic learning in [Schulz 89]. In [Rajski 90], the authors introduced the use of 16-valued algebra and reduction lists to determine the necessary node assignments in ATPG. Transitive closure procedure on implication graph was proposed in [Chakradhar 93] to improve indirect implications. A more complete implication engine based on recursive learning was proposed by Kunz *et al.* [Kunz 92]. However, as the depth of recursion is increased the time to compute the implications increases exponentially. Hence, to keep the computation time within reasonable limits, the recursion depth was bound to low values. A graphical representation of the implication relations was proposed by Zhao *et al.* in [Zhao 01] and the concept of indirect implications based on transitivity of implications, along with extended backward implications was used to enhance the power of implication engine.

1.2 Thesis Outline

An outline of the rest of the Thesis is as follows:

- Chapter 2 gives the necessary definitions and the terminology used. It provides details pertaining to static logic implications (both combinational and sequential), the basics of a SAT-solver, an overview of SAT-based Combinational Equivalence Checking (CEC) and Bounded Model Checking (BMC), and finally it briefly discusses the untestable faults and techniques used for their identification.
- Chapter 3 describes how the combinational static logic implications can be applied to enhance SAT-based CEC, yielding more than an order of magnitude speedup over the conventional approach
- Chapter 4 describes a novel approach of using sequential static logic implications to boost the performance of SAT-based BMC. The approach is the first of its kind to use implication relationships spanning time-frame boundaries for inducing signal correlations into the original CNF database.
- Chapter 5 presents a novel global learning technique resulting in a new class of implications, termed as e*xtended forward implications*. We show through experimental results how these non-trivial implications can be applied to increase the identification of untestable faults in combinational circuits.
- Chapter 6 presents a suite of Lemmas and Theorems for the simplification of a generic CNF formula, and compares the performance our new tool with the previous approaches.
- Chapter 7 concludes the work with an overview and presents some recommendations for future work.

CHAPTER 2

PRELIMINARIES

2.1 Static Logic Implications

Static implications are obtained by setting each gate in the Boolean circuit to logic value 1 and 0, and analyzing the result of propagating these values throughout the circuit. The following terminology is used:

- (N, v, t): Assign logic value v to gate N in time frame t, where $v \in \{0, 1\}$. For combinational circuits, t is equal to 0, and is dropped from the expression, i.e. if $t = 0$, $(N, v, t) \equiv (N, v)$.
- $(N, v) \rightarrow (M, w, t)$: Assigning logic value v to gate N implies gate M would be assigned logic value w in time frame t.
- *impl* $[N, v, t]$: Set of all implications resulting from assigning logic value v to gate N in time frame t. Again, for combinational circuits $t = 0$, and is dropped.
- *contrapositive law* [Schulz 88]: If $(N, v) \rightarrow (M, w, t)$, then the contrapositive law states that $(M, w') \rightarrow (N, v', -t)$, where w' and v' are the complementary values of w and v, respectively. This property can be used to identify additional (possibly non-trivial) implications.
- *impossible/constant nodes*: If $(M, w) \rightarrow (N, v, t)$ and $(M, w) \rightarrow (N, v', t)$ or if $(M, w) \rightarrow (M, w')$, then (M, w) is impossible, i.e. gate M would never be able to

acquire value *w* and would be a constant with value *w'* (for clear understanding refer to Figure 2.2 and the text under direct implications)

- *transitive law:* If $(M, w) \rightarrow (N, v, t_1)$ and $(N, v) \rightarrow (P, u, t_2)$, then the transitive law states that $(M, w) \rightarrow (P, u, t_1 + t_2)$.
- *fanins:* All gates that drive gate *N*
- *fanouts:* All gates that are driven by gate *N*
- *target gate:* The gate whose implications are being computed by setting it to logic value 0 or 1.
- *unjustified gate:* A gate G that has a specified output signal or at least one specified input signal; if the output signal is specified, it is not determined by its inputs/fanins. And if any of the inputs/fanins are specified, they do not determine the gates output value.
- *unjustified output specified gate:* Subset of unjustified gates whose output value is specified, but is not determined by its inputs/fanins.
- *controlling value:* A logic value at any of the fanins which can determine the gate's output value (see Table 2.1 for controlling values of different gate types).
- *inversion value:* If the output of the gate is inverted as in the case of NOT, NAND, and NOR gates, the inversion value is 1; otherwise 0 (see Table 2.1 for inversion values of different gate types).

The static logic implications are made up of direct, indirect and extended backward implications. Direct implications can be easily determined whereas indirect and extended backward implications [Zhao 97, Zhao 01] are non-trivial, and their discoveries

require combination of simulation, transitive law and contrapositive law [Schulz 88]. The mathematical definitions of direct, indirect and extended backward implications are given below and the concepts illustrated through the example circuit shown in Figure 2.1.

2.1.1. Direct Implications

Direct implications of a gate *G* consist of implications associated with the gates *driving* and *driven* by *G*. Such implications are easily computed by traversing through the immediate fanins and fanouts of the gate. The direct implications are of two types: 1) direct forward implications, and 2) direct backward implications. To compute direct forward implications, a controlling value of *c* at any of the fanins implies a value of *c XOR i* at the gate output, where *i* is the inversion value of the gate. Table 2.1 gives the controlling value (*c*), the non-controlling value (*nc*) and the inversion value (*i*) for different gates. Note that the non-controlling (*nc*) value is just the complement of the controlling value (*c*). Also, note that XOR and XNOR gates do not have any controlling or non-controlling value. Similarly, to compute direct backward implications, a value of *nc XOR i* at the output implies a value of *nc* at all the fanins.

Table 2.1 Controlling, Non-Controlling and Inversion values for various gates

Gate	Controlling value (c)	Non Controlling value (nc)	Inversion value (i)
AND	0	1	0
NAND	0	1	1
OR	1	0	0
NOR	1	0	1

Consider the example circuit in Figure 2.1. Here, *e* represents an OR gate, *f, h* and *k* are NAND gates, *i* and *m* are AND gates, *g* is a NOT gate and *j* is an XOR gate. Now consider gate *f*. When we assert a logic value 0 on its output, the *direct forward implications* are (*g, 1*) and (*h, 1*). Similarly, the *direct backward implications* are (*e, 1*) and (*c, 1*). Therefore, *impl* [*f, 0*] = {(*f, 0*), (*g, 1*),(*h, 1*), (*e, 1*), (*c, 1*)}. These direct implications are shown in Figure 2.1.

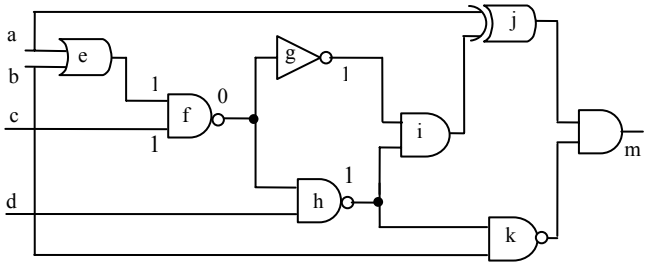


Figure 2.1 Example Combinational Circuit

An example circuit showing how direct implications lead to constant nodes is shown in Figure 2.2. Here, *impl* [*c, 0*] = {(*c, 0*), (*a, 1*), (*b, 1*)}, *impl* [*b, 1*] = {(*b, 1*), (*a, 0*)}, *impl* [*b, 0*] = {(*b, 0*), (*a, 1*), (*c, 1*)}, *impl* [*a, 1*] = {(*a, 1*), (*b, 0*)} and *impl* [*a, 0*] = {(*a, 0*), (*b, 1*), (*c, 1*)}. Hence, taking the transitive closure of implications of (*c, 0*) we get *impl* [*c, 0*] = {(*c, 0*), (*a, 1*), (*b, 0*), (*c, 1*) (*a, 0*), (*b, 1*)}. Since *impl* [*c, 0*] contains both (*a, 1*) and (*a, 0*), therefore (*c, 0*) is an impossible assignment and *c* should be a constant with logic value *1*. We can also interpret this in a different way. Since, *impl* [*c, 0*] contains (*c, 1*) i.e. (*c, 0*) → (*c, 1*) therefore *c* is a constant with logic value 1.

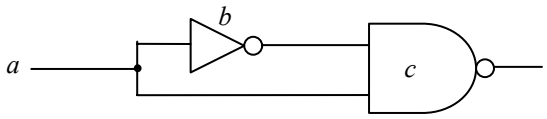


Figure 2.2 Example Circuit illustrating Constant/Impossible nodes

2.1.2 Indirect Implications

The indirect implications of gate *N* set to value *v* are computed by plugging the gate values pertaining to all its direct implications onto the circuit, and performing logic simulation. All gates whose output value changes from a don't-care to logic *0* or *1,* form the indirect implications of (*N, v*).

Mathematically, *impl* [*N, v*] ≡ *impl* [*N, v*] ∪ [*LogicSimulate* (*impl* [*N, v*])].

Here *LogicSimulate* () refers to performing logic simulation with the implications plugged onto the circuit.

Consider the direct implications of (*f, 0*) shown in Figure 2.1. We see that (*g, 1*) or (*h, 1*) individually do not imply anything on gate *i*. However, together they imply (*i, 1*). Therefore, (*f, 0*) → (*i, 1*) is an indirect implication, and can be computed by simply logic simulating the list *impl* [*f, 0*]. Thus, the new list for *impl* [*f, 0*] = {(*f, 0*), (*g, 1*), (*h, 1*), (*e, 1*), (*c, 1*), (*i, 1*)}. These indirect implications have been used in the past by the name of *global implications* and *non-local implications*. Schulz *et al*. in [Schulz 88] utilized these non-local implications to improve the performance of ATPG engine and later Larrabee and Stephan *et al.* in [Larabee 92] and [Stephan 96], respectively, used them for combinational test generation in a SAT framework.

2.1.3 Extended Backward Implications

The extended backward implications were first introduced by Zhao *et al*. in [Zhao 97]. These implications are computed by considering (1) the *target gate* and (2) the *unjustified output specified gates* in the implication list of the target gate.

Let $(G, v) \in impl\ [N, v]$, and suppose gate G has p inputs, among which m inputs $(l_1, ..l_m)$ are unspecified. Here N is the *target gate* and G is the *unjustified output specified gate.*

Case 1: G is an AND gate:

If $(G, 0) \in impl\ [N, v]$ and $(l_j, 0) \notin impl\ [N, v]$, $(j = 1,2,....p)$, then

$$impl[N, v] \equiv impl[N, v] \cup [\cap_{i=1}^{m} LogicSimulate(impl[N, v] \cup impl[l_i, 0])]$$

The above mathematical formulation states that if the implication set of (N, v) contains an AND gate G which is unjustified output specified (i.e. it has an output value of 0 which is not determined by the value of its fanins), then the common set of implications obtained by setting each of the unspecified fanins to 0 under the current assignment of (N, v), will be appended to the implication set of (N, v)

Case 2: G is an OR gate:

If $(G, 1) \in impl[N, v]$ and $(l_j, 1) \notin impl[N, v]$, $(j = 1,2,....p)$, then

$$impl[N, v] \equiv impl[N, v] \cup [\cap_{i=1}^{m} LogicSimulate(impl[N, v] \cup impl[l_i, 1])]$$

The above mathematical formulation states that if the implication set of (N, v) contains an OR gate G which is unjustified output specified (i.e. it has an output value of 1 which is not determined by the value of its fanins), then the common set of implications obtained by setting each of the unspecified fanins to 1 under the current assignment of (N, v), will be appended to the implication set of (N, v)

In the same way, extended backward implications can be computed for *NAND* and *NOR gates.*

Case 3: G is a 2-input XOR gate:

If $(G, 1) \in impl[N, v]$ and both inputs l_0 and l_1 are unspecified, then,

$impl\ [N, v] \equiv impl\ [N, v] \cup \{\ LogicSimulate\ (impl\ [N, v] \cup impl\ [l_0, 0] \cup\ impl\ [l_1, 1]) \cap$

$LogicSimulate\ (\ impl\ [N, v] \cup impl\ [l_0, 1] \cup impl\ [l_1, 0])\ \}$

The above mathematical formulation states that if the implication set of (N, v) contains an XOR gate G which is unjustified output specified (i.e. it has an output value of 1 which is not determined by its fanins), then the common set of implications obtained by setting its two fanins to logic value 0 and 1 and then to 1 and 0, respectively, under the current assignment of (N, v), will be appended to the implication set of (N, v)

If $(G, 0) \in impl\ [N, v]$ and both inputs l_0 and l_1 are unspecified then,

$impl\ [N, v] \equiv impl\ [N, v] \cup \{\ LogicSimulate\ (impl\ [N, v] \cup impl\ [l_0,0] \cup\ impl\ [l_1, 0]) \cap$

$LogicSimulate\ (\ impl\ [N, v] \cup impl\ [l_0, 1] \cup impl\ [l_1, 1])\ \}$

The above mathematical formulation states that if the implication set of (N, v) contains an XOR gate G which is unjustified output specified (i.e. it has an output value of 0 which is not determined by its fanins), then the common set of implications obtained by setting both the fanins to logic value 0 and then to 1, under the current assignment of (N, v), will be appended to the implication set of (N, v)

Case 4: G is a 2-input XNOR gate:

If $(G, 0) \in impl\ [N, v]$ and both inputs l_0 and l_1 are unspecified then,

$impl\ [N, v] \equiv impl\ [N, v] \cup \{\ LogicSimulate\ (impl\ [N, v] \cup impl\ [l_0, 0] \cup\ impl\ [l_1, 1]) \cap$

$LogicSimulate\ (\ impl\ [N, v] \cup impl\ [l_0, 1] \cup impl\ [l_1, 0])\ \}$

If $(G, 1) \in$ *impl* $[N, v]$ and both inputs l_0 and l_1 are unspecified then,

impl $[N, v] \equiv$ *impl* $[N, v] \cup \{$ *LogicSimulate* (*impl* $[N, v] \cup$ *impl* $[l_0, 0] \cup$ *impl* $[l_1, 0]) \cap$ *LogicSimulate* (*impl* $[N, v] \cup$ *impl* $[l_0, 1] \cup$ *impl* $[l_1, 1]) \}$

Since we deal with miter circuits for combinational equivalence checking (see Section 5.3), the extended backward implications pertaining to XOR/XNOR gates help to identify many non-trivial implications, which in turn play an important role in proving the equivalence of the two circuits.

To illustrate the concept of extended backward implications, consider again the example circuit of Figure 2.1. We saw that after computing indirect implications, the list *impl* $[f, 0] = \{(f, 0), (g, 1), (h, 1), (e, 1), (c, 1), (i, 1)\}$. The implication list of $(f, 0)$ contains $(e, 1)$ and the OR gate e is unjustified output specified. Now justifying $e = 1$, by setting the fanin $a = 1$ yields XOR gate $j = 0$ and $j = 0 \rightarrow m = 0$. Setting the fanin $b = 1$ results in NAND gate $k = 0$ and $k = 0 \rightarrow m = 0$. Thus, if the OR gate e is justified by any of the fanins (i.e. a or b) under the condition $(f, 0)$, we get a common implication $m = 0$. Therefore, $f = 0 \rightarrow m = 0$ is an extended backward implication of $(f, 0)$, and is appended to the list *impl* $[f, 0]$. These extended backward implications help to identify the *hard-to-find implications,* and hence are effective for various applications such as capturing additional untestable faults [Zhao 97, Zhao 01, Hsiao 02].

2.1.4 Implication Graph

In general, the total number of implications associated with the entire circuit can be exponential in the size of the circuit. Thus, a memory efficient technique must be used

to store the implications associated with each gate. We use a directed-graph based representation proposed by Zhao *et al*. in [Zhao 01] for storing the implications. This representation has the advantage of being used for sequential circuits without suffering from the problem of memory explosion. For a given circuit with K gates, the total number of nodes in this graph is $2K$, since each gate can take on a logic value of 0 or 1. A directed edge between two nodes (i.e. from node a to node b) represents an implication between the two nodes (i.e. $a \rightarrow b$). The weight associated with an edge represents the relative time frame associated with the implication. It should be noted that the time-frames are bounded by D flip-flops and the current time-frame is always time-frame *0.* When an implication propagates across a D flip-flop, the time frame is incremented or decremented accordingly. Also, by representing the sequential implications as a graph, transitive closure of a node can be easily obtained using the *depth-first search* technique. Another advantage is that whenever a new indirect or extended backward implication is computed, its contrapositive implication (from contrapositive law) can be immediately added to the implication graph. Figure 2.3 shows the example sequential circuit and Figure 2.4 shows its partial implication graph.

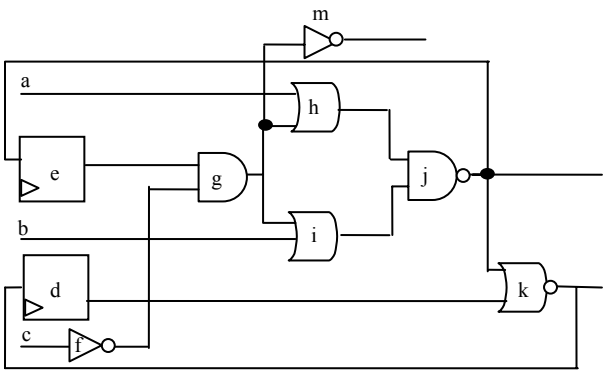


Figure 2.3 Example Sequential Circuit

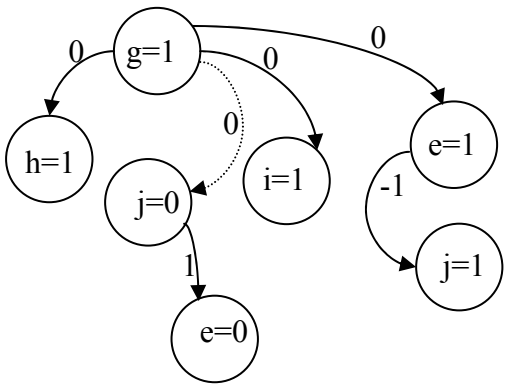


Figure 2.4 Partial Implication Graph for Sequential Circuit in Figure 2.3

In Figure 2.4, a directed edge between two nodes (e.g. from *node g=1* to *node h=1*) represents an implication between these two nodes (i.e. (*g, 1*) → (*h, 1*)). As said earlier, the weight on the edge represents the relative time frame associated with the implication. Therefore, *e=1* implies *j=1* in time frame *-1*, and is represented as *(e, 1) → (j, 1, -1)*. This nice representation allows for implications to span multiple time frames without explicit unrolling of the circuit. For instance, because *(g, 1) → (e, 1),* and *(e, 1) → (j, 1, -1)* as shown in the implication graph in Figure 2.4, *(g, 1) → (j, 1, -1)* by transitive property. In general, the transitive law helps to deduce implication relations with edge weights ranging from *-n* to *+n* (*n* being a whole number). However, in case of a loop, *n* can be *infinity*. In our implementation, we restrict this value of *n* and make it user-specified. We call this *n* as *sequential implication depth* or *maximum edge weight.* Note that the cross time-frame implications are obtained without having to unroll the circuit and are quickly determined.

2.2 Boolean Satisfiability (SAT)

The Boolean Satisfiability (SAT) problem consists of determining a satisfying variable assignment *V*, for a Boolean function *f*, or determining that no such *V* exists. SAT belongs to the class of NP-complete problems whose algorithmic solutions are currently believed to have exponential worst case complexity. The state-of-the-art SAT solvers [Moskewicz 01, Goldberg 02a, Ryan 03] are descendants of the DPLL-algorithm [Davis 62] and are usually based on the Conjunctive Normal Form (CNF) also commonly known as Product of Sum (POS) form. This form consists of the logical AND (conjunction) of one or more *clauses*, such that each *clause* is a logical OR (disjunction) of one or more *literals*. A *literal* may be a boolean variable in its true or complement

form. For the CNF formula to be *satisfied,* each of the individual clauses should be satisfied (*sat*). Each of these clauses are also called *implicates* of the CNF formula. While trying to satisfy this given CNF formula, the SAT-solver makes decisions based on a given set of variable selection heuristics [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97]. It learns dynamically from the conflicts encountered during the search and generates conflict-induced clauses [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97] that can subsequently constrain the search.

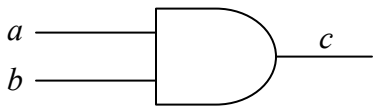
The following terms are used:

- *unit-clause rule:* If a clause has n literals and $n - 1$ of its literals have been assigned to logic value 0 by the current state of decision assignments, then the unassigned literal should take on logic value 1 for the CNF formula to be satisfiable. This literal is called as *unit literal* or *implied value*.
- *Boolean Constraint Propagation* (*BCP*) [Moskewicz 01, Silva 99a]: Applying the unit-clause rule repeatedly until no more unit literals can be obtained.
- *BCP* (x, v): Set of values implied by performing BCP with x assigned to logic value v.

2.2.1 CNF Formula Derivation for Primitive Gates

In this subsection we will show how the CNF formula can be derived for the primitive gate types. The CNF formula for the circuit is then the conjunction of the CNF formula for all the gates in the circuit. The readers are referred to [Larrabee 92] for an in-depth study on CNF formula derivation.

1. AND gate



$c=ab$

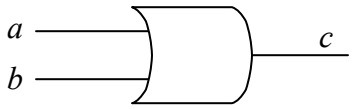
The CNF formula for the AND gate is:

$(c \rightarrow ab)\ (ab \rightarrow c)$

$\Rightarrow \quad (\neg c \vee ab)\ (\neg ab \vee c)$

$\Rightarrow \quad \mathbf{(\neg c \vee a)(\neg c \vee b)\ (\neg a \vee \neg b \vee c)}$ ….using DeMorgan's and Distributive laws

2. OR gate



$c = a \vee b$

The CNF formula for the OR gate is:

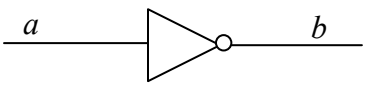
$(c \rightarrow a \vee b)\ (a \vee b \rightarrow c)$

$\Rightarrow \quad (\neg c \vee a \vee b)\ (\neg(a \vee b) \vee c)$

$\Rightarrow \quad (\neg c \vee a \vee b)\ (\neg a\ \neg b \vee c)$

$\Rightarrow \quad \mathbf{(\neg c \vee a \vee b)\ (\neg a \vee c)(\ \neg b \vee c)}$ ….using DeMorgan's and distributive laws

3. NOT gate



$a = \neg b$

The CNF formula for the NOT gate is:

$(a \rightarrow \neg b)\ (b \rightarrow \neg a)$

$\Rightarrow \quad \mathbf{(\neg a \vee b)\ (\neg b \vee a)}$

Similarly, the CNF formula can be derived for other gate types as well.

The CNF formula for a Boolean circuit is then the conjunction of the CNF formula for the different gates constituting that circuit.

2.2.2 Simple Algorithm for SAT

A naive branch and bound algorithm for solving a SAT instance is shown below:

1. Pick a variable *v*
2. Set *v=0* or *1*
3. Propagate *v* to the CNF formula by applying unit clause rule repeatedly (also called *Boolean Constraint Propagation*)
4. If any clause evaluates to 0, *backtrack*
5. Repeat until all the clauses get satisfied or a satisfying assignment is not found.

It should be noted that this branch and bound search procedure implicitly traverses the space of 2^n possible binary assignments.

To illustrate how the above SAT algorithm works, consider the example circuit shown in Figure 2.5. The CNF formula Φ for this example is given by:

$\omega 1 = (\neg c \vee a)$, $\omega 2 = (\neg c \vee b)$, $\omega 3 = (\neg a \vee \neg b \vee c)$ ..... for AND gate *c*

$\omega 4 = (d \vee a)$, $\omega 5 = (d \vee b)$, $\omega 6 = (\neg a \vee \neg b \vee \neg d)$ ..... for NAND gate *d*

$\omega 7 = (e \vee \neg c)$, $\omega 8 = (e \vee \neg d)$, $\omega 9 = (c \vee d \vee \neg e)$ ..... for OR gate *e*

Let our objective be *e=0*. So we make the decision *e=0* and propagate this value to the CNF formula. Applying the unit clause rule to the clauses ω7 and ω8, yields *c=0* and *d=0,* respectively. Now propagating *d=0* and applying unit clause rule to ω4 and ω5 yields *a=1* and *b=1,* respectively. However, *a=1*, *b=1* and *c=0* together cause the

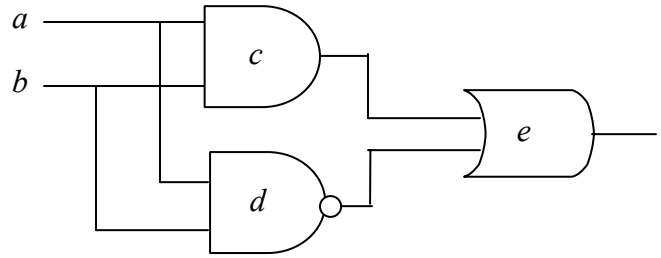


Figure 2.5 Example Circuit illustrating the SAT Algorithm

clause ω3 to evaluate to *0* and hence results in a conflict. Thus, we backtrack to the most recent decision, which in this case is *e=0*. *e* when set to opposite value *1* violates the objective (i.e. *e=0*). Hence, a solution does not exist for this case. However, if our objective would have been *e=1,* there would have existed a solution whereby *e=1, c=1, d=0, a=1,* and *b=1*. For an in-depth study on SAT-algorithms and the recent advancements in SAT-solvers the readers are referred to [Goldberg 02a, Moskewicz 01, Silva 99a, ZhangH 97].

2.3 SAT-based Combinational Equivalence Checking (CEC)

Combinational Equivalence Checking (CEC) requires that the implementation circuit should be exactly equivalent to the specification circuit (*golden model*). In other words, for every possible input pattern applied, the implementation circuit should yield the same outputs as the specification circuit. The framework for Combinational Equivalence Checking (CEC) is shown in Figure 2.6 and is called as the *miter circuit*. The specification circuit (can be an un-optimized version) is denoted by C_S and the implementation circuit (can be an optimized version) is denoted by C_I. We see from Figure 2.6 that the primary inputs (PIs) of the two circuits are tied together and the corresponding primary outputs are XORed which in turn are fed to an OR gate. If the two

circuits are exactly equivalent, then for every possible input pattern the OR gate output should not experience a logic *1*. In other words, the OR gate output should be a constant *0*. In a SAT-based framework, we generate the CNF formula for the miter circuit under verification and ask the SAT-solver to satisfy the OR gate output to logic *1* (this is our *objective*). If the two circuits C_S and C_I are exactly equivalent then the corresponding CNF instance will be unsatisfiable, otherwise satisfiable.

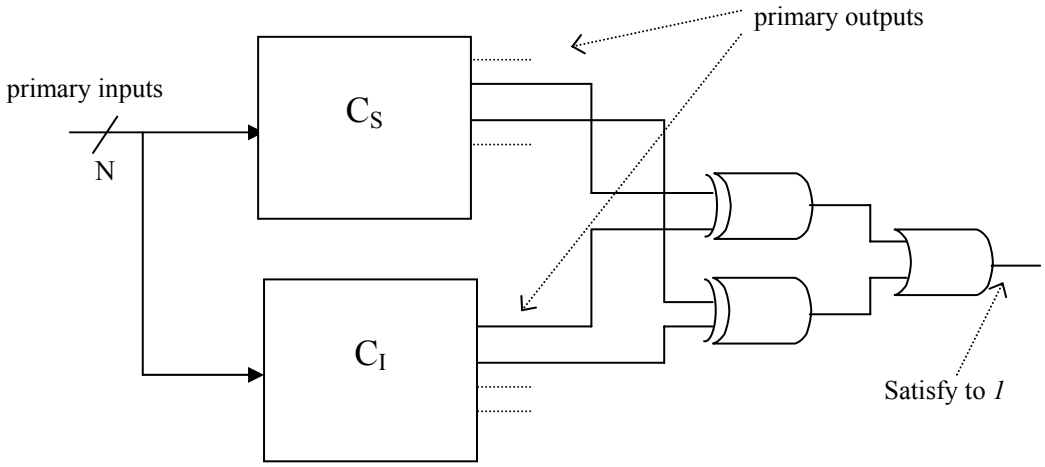


Figure 2.6 Combinational Equivalence Checking (CEC) Framework

In Chapter 3, we show how we have been able to improve the performance of SAT-based Combinational Equivalence Checking (CEC) by utilizing the static logic implications spanning the entire miter circuit.

2.4 SAT-based Bounded Model Checking (BMC)

In SAT-based Bounded Model Checking (BMC), given a property of the Finite State Machine (FSM) model, the SAT-solver tries to determine if the property holds in the bounded length *k*. These properties which are indispensable for the proper functionality of the design are mapped onto the flip-flops or the state variables. If the

property is violated by the given implemented model, a counter-example or a trace is generated.

Two kinds of properties that are usually checked using Bounded Model Checking belong to the CTL class [Clarke 86]. These are:

1. safety property - It denotes that a certain condition crucial for the proper functioning of the design must not be violated at any time instance, i.e. *something bad will never happen*. As a CTL formula it is represented by *EF*(*f*), which means that starting form the initial state there exists a path that leads to a state where *f* holds. For example, here *f* could denote an illegal state and if *EF*(*f*) is true/satisfied, a counterexample will be generated, implying that the processor enters an illegal state (which is bad).
2. liveness property – It denotes that the desired or necessary system condition will eventually happen i.e. *something good will eventually happen*. As a CTL formula it is represented by *EG*(*f*), which means that starting form the initial state there exists a path such that *f* holds in every state. For example, a liveness property could be such that the processor always returns to the fetch state s_f. This could be written in CTL form as $EG(\neg s_f)$, which if true would imply that along some execution path for all future states the fetch state is not reached. The counterexample so generated will be an infinite sequence of states which does not include the fetch state.

To perform BMC in the bounded length *k,* the sequential circuit is firstly unrolled into *k* time-frames. While unrolling, the D flip-flops in the first time frame are treated as Pseudo Primary Inputs (PPIs); for subsequent time-frames they are converted into buffers and fed to the combinational portion of the sequential circuit. Finally, for the last time-

frame these flip-flops are treated as Pseudo Primary Outputs (PPOs). After performing sequential unrolling, Bounded Model Checking (BMC) Circuitry called as *Monitor Circuit* [Boppana 99] is constructed, corresponding to the property to be verified. The Bounded Model Checking Framework is shown in Figure 2.7. A CNF database is built for this transformed circuit (Figure 2.7) and the SAT solver is asked to satisfy the *Monitor Circuit* output to logic *1*. For example, consider that the sequential circuit under

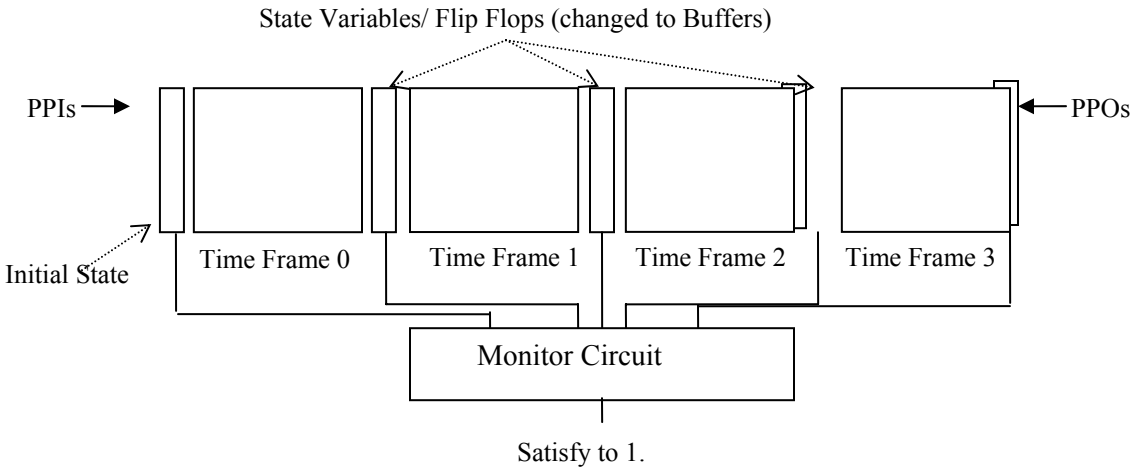


Figure 2.7 Bounded Model Checking (BMC) Framework

verification has 6 flop-flops ($S_1S_2S_3S_4S_5S_6$). Suppose the starting/initial state is (101010) and the safety property *EF*(*0X0X10*) needs to be verified. A monitor circuit is constructed such that it evaluates to logic *1* if the target state (*0X0X10*) can be reached in *any* of the *k* time frames. The starting state (*101010*) is added to the existing CNF database as a constraint (in the form of unit clauses).

The efficiency of the BMC formulation shown in Figure 2.7 depends on the complexity of the circuit-under-verification, as well as the underlying SAT-solver. In Chapter 4, we show how we have been able to improve the performance of SAT-based

Bounded Model Checking (BMC) using sequential logic implication relationships, spanning time-frame boundaries.

2.5 Untestable Faults and Techniques for their Identification

Untestable faults are faults for which there exists no test sequence that can both excite the fault-effect and propagate it to a primary output (PO). In combinational circuits, untestable faults result from redundant logic in the circuit, while in sequential circuits, untestable faults can also result from unreachable states. The current state-of-the art automatic test pattern generators (ATPG) spend a lot of computational effort in attempting to generate a test pattern for the detection of such untestable faults, before aborting on them, or identifying the faults as untestable (given enough time).Thus, the performance of fault-oriented tools such as test-pattern-generators and fault-simulators can be enhanced if knowledge of untestable faults is available *a priori*. There are additional indirect benefits of untestable fault identification because such faults can have other detrimental effects. The presence of untestable faults can potentially prevent the detection of other faults in the circuit. Untestable faults in the form of redundancies increase the chip area, and may also cause increase in power consumption and propagation delays through the circuit. Also, untestable faults may result in unnecessary yield loss during IDDQ or full-scan mode of testing.

Techniques used for untestable fault identification can be broadly classified into fault-oriented methods [Agarwal 95, Reddy 99, Peng 00], and fault independent techniques [Iyer 96a, Iyer 96b, Hsiao 02] based on conflict analysis. Generally, ATPG based methods [Agarwal 95, Reddy 99] outperform fault-independent methods for

smaller circuits; however, the computational complexity of branch-and-bound algorithms (ATPG) render them impractical for large circuits. Conflict based analysis have thus been researched and improved over the years. In [Iyer 96a] a fault independent redundancy identification technique named FIRE was introduced to identify untestable faults that require a conflict on a single line as a necessary condition for their detection. FIRES [Iyer 96b] was introduced as an extension of FIRE to identify untestable faults in sequential circuits without explicit search. The MUST algorithm proposed in [Peng 00] was built over the framework of FIRES as a fault-oriented approach to identify untestable faults. However, the memory requirement for MUST can be exponential. Recently, Hsiao in [Hsiao 02] presented a fault independent technique to identify untestable faults by utilizing impossible value combinations locally around each Boolean gate in the circuit netlist. The technique was shown to be highly effective since it increased the number of untestable faults by a huge margin, with a little increase in computation effort. Since the success of such fault independent techniques depends upon the implication engine these techniques utilize, it's important to have as large an implication set associated with each line as possible.

2.5.1. Redundancy Identification using Single-Line Conflict (FIRE algorithm)

The underlying concept behind single-line conflict analysis [Iyer 96a] (also called as the FIRE algorithm) lays in the fact that faults that require a conflict on a single line as a necessary condition for their detection are untestable. In this analysis, for every gate g (or stem s), the following two sets are computed:

Set_0: Set of faults untestable with $g = 0$. The faults in this set require $g = 1$ to be

testable.

Set_1: Set of faults untestable with $g = 1$. The faults in this set require the assignment $g = 0$ to be testable.

The set of untestable faults is then the intersection of the two sets, Set_0 and Set_1. For better understanding, consider the example circuit shown in Figure 2.8.

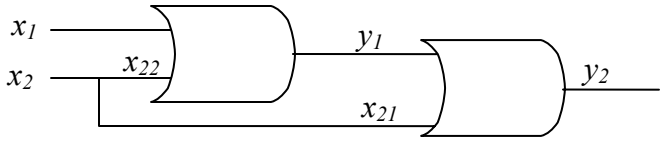


Figure 2.8. Example circuit illustrating FIRE algorithm

Consider stem x_2 in the example circuit shown in Figure 2.8. In the discussion that follows, assume that *g/v* denotes the fault *g stuck at v*. The set of faults that would become untestable (unexcitable and unobservable) when *x2 = 0* are:

$Set_0 = \{ x_2/0, x_{22}/0, x_{21}/0\}$.

Similarly, the set of faults that would become untestable when *x2 = 1* are:

$Set_1 = \{ x_2/1, , x_{22}/1, x_{21}/1, y_1/1, y_2/1, x_1//0, x_1/1, x_{21}/0, , y_1/0\}$

Thus, $Set_0 \cap Set_1 = \{ x_{21}/0 \}$. Hence, the fault $x_{21}/0$ is untestable according to single-line-conflict analysis, since it requires conflicting assignments on x_2 for its detection.

2.5.2 Redundancy Identification using Multi-Line Conflict ([Hsiao 02])

In Chapter 5, we introduce a novel global learning technique termed *as extended forward implications* which enhances the power of existing implication engine, and hence can be used to increase the identification of untestable faults. To show the efficacy of

extended forward implications in identifying additional untestable faults, we implemented the Impossible Value Combination (IVC) algorithm proposed in [Hsiao 02]. The IVC algorithm which is built on top of the FIRE algorithm is described below:

1. Construct the Implication Graph for the circuit under test (using direct, indirect and extended backward implications)
2. For each line *l* in the circuit

   Identify all untestable faults using the single line-conflict FIRE algorithm
3. // The Impossible Value Combination (IVC) Algorithm

   For each gate *G* in the circuit

   SIV= Set of impossible value combinations for gate *G*

   a. $i = 0$

   b. for each value assignment ($a = v$) in *SIV*

   set_i = faults requiring $a = \neg v$ to be detectable

   $i = i + 1$

   c. untestable_faults = untestable_faults $\cup$ ($\cap_{\forall i}$ set_i)
4. Stop

Each gate type *G* has exactly *one* impossible value combination wherein all its fanins are at *nc* value and the gate's output is at a *c XOR i* value, where *nc, c* and *i* are the non-controlling, controlling and inversion values of the gate. For example in case of a 3-input NAND gate *x* with fanins *p, q* and *r*, the set $SIV = \{ x = 1, p = 1, q = 1, r = 1 \}$

In Chapter 5 we show how the enhanced implication engine (strengthened by *extended forward implications*) when used with the IVC algorithm, helps to identify additional untestable faults.

CHAPTER 3

ENHANCING COMBINATIONAL EQUIVALENCE CHECKING (CEC) IN A SAT-BASED FRAMEWORK

In this chapter, we propose a technique for improving SAT-based Combinational Equivalence Checking (CEC). In our approach, unlike [Lu 03a, Novikov 03], we statically and efficiently identify useful non-trivial relations among signals (variables) over the *entire* miter circuit. We then augment the existing CNF formula by adding these relations as clauses, before the SAT solver starts. Instead of working on the CNF formula as in [Bacchus 02, Bacchus 03, Lynce 03, Gelder 93], we work on the circuit netlist for inferring additional clauses. The pre-processing step quickly builds the implication graph [Zhao 01] for the miter-circuit under verification. The resulting indirect and extended backward implications help us to deduce unit literals (variables with constant 0 or 1 logic value), equivalent literals and other non-trivial implication relations among the CNF variables. The non-trivial implication relationships are converted into two-literal clauses, which are added to the CNF database. These added clauses prune the search space and provide correlation among different variables, which enhances the Boolean Constraint Propagation [Moskewicz 01, Silva 99a]. Two state-of-the-art SAT solvers are used in our experiments: *BerkMin* [Goldberg 02a] and *Siege* [Ryan 03]. Experimental results for combinational circuit equivalence checking show that our proposed method is

independent of the underlying SAT-solver, and we achieve more than an order of magnitude speedup over the conventional approach. Also, comparison with hyper binary resolution [Bacchus 02, Bacchus 03] suggests that our proposed technique is much more powerful and the resulting non-trivial clauses are difficult to obtain using the hyper resolution approach. These new clauses when added to the original CNF formula reduce the SAT instance complexity significantly.

3.1 Application of Static Implications to SAT-based CEC

When a circuit netlist is converted into its equivalent CNF-form, the resulting formula is devoid of global structural information. Also, the topological ordering among the signals is lost. All the internal signals in the original circuit become primary inputs (variables) in the two-level OR-AND CNF formula. As a result, the SAT-solver heuristically picks up a variable for decision, without having much information about its impact on future decisions. For example, successive decisions on two different variables might be correlated in some way, but due to absence of global relationships, these variables may be assigned values that may eventually lead to a conflict in the future. In our approach, we try to induce structural relationships into the CNF formula of the miter circuit under verification, such that conflicts are either completely avoided or can be deduced early in the decision process. We first compute the static implications on the circuit netlist, and then convert these implications into clause form. These clauses when added to the original CNF formula induce signal correlation among the variables, which in turn enhances the SAT solver performance.

3.1.1 Enhanced Boolean Constraint Propagation (BCP)

Consider the example circuit of Figure 2.1 in Chapter 2. Its CNF formula is shown below. The CNF formula derivation is straightforward and has been discussed in Chapter 2, under sub-section 2.2.1.

$$
\begin{array}{l}
(\neg a \vee e)\ (\neg b \vee e)\ (\neg e \vee a \vee b)\ (f \vee e)\ (f \vee c) \\
(\neg f \vee \neg e \vee \neg c)\ (\neg f \vee \neg g)\ (f \vee g)\ (\neg f \vee \neg d \vee \neg h)\ (f \vee h) \\
(d \vee h)(g \vee \neg i)\ (h \vee \neg i)\ (\neg g \vee \neg h \vee i)\ (\neg j \vee a \vee i) \\
(\neg j \vee \neg a \vee \neg i)\ (j \vee \neg a \vee i)\ (j \vee a \vee \neg i)\ (h \vee k)\ (b \vee k) \\
(\neg h \vee \neg b \vee \neg k)\ (j \vee \neg m)\ (k \vee \neg m)\ (m \vee \neg j \vee \neg k)
\end{array}
$$

In this CNF formula, the clauses $(\neg a \vee e)(\neg b \vee e)(\neg e \vee a \vee b)$ represent the OR gate e, $(f \vee e)(f \vee c)(\neg f \vee \neg e \vee \neg c)$ represent the NAND gate f, $(\neg f \vee \neg g)(f \vee g)$ correspond to NOT gate g, $(\neg j \vee a \vee i)(\neg j \vee \neg a \vee \neg i)(j \vee \neg a \vee i)(j \vee a \vee \neg i)$ correspond to XOR gate j and so on.

Now, let us suppose that the SAT solver heuristically makes the first decision $i = 0$. On assigning $i = 0$ and performing Boolean Constraint Propagation (BCP), *4* clauses are satisfied and no unit clauses are obtained. However, from our implication engine, we know that $f = 0 \rightarrow i = 1$, and by contrapositive law $i = 0 \rightarrow f = 1$. The two-literal clause corresponding to this implication is $(i \vee f)$. If we add this clause beforehand to the original CNF formula, setting $i = 0$ will imply $f = 1$ immediately, which in turn will imply $g = 0$. Therefore, learning the information $i = 0 \rightarrow f = 1$, helps us to satisfy a total of *10 clauses* instead of satisfying only *4*. This is illustrated in Figure 3.1. Thus, we see that addition of a single clause results in significant improvement in BCP. In our

approach, we add large number of such clauses (corresponding to indirect and extended backward implication) which in turn improves the BCP considerably.

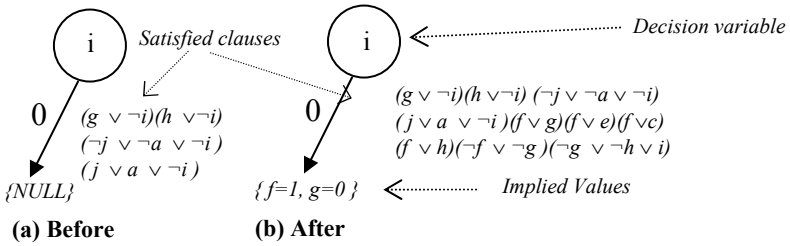


Figure 3.1 Implied Values and Satisfied Clauses in the CNF formula, before and after adding the Clause $(i \vee f)$

3.1.2 Identification of Equivalent/Complement Literals

The basis of Combinational Equivalence Checking (CEC) is to identify equivalent signals in the two circuits incrementally, proceeding from the primary inputs towards the primary outputs. In SAT-based CEC, identification of such equivalent signals helps to reduce the problem complexity; a decision on one of the signals in the equivalent pair implies a value on the other corresponding signal, which in turn enhances the BCP and reduces the number of decisions required to prove the satisfiability/unsatisfiability of the CNF formula. The implication graph that we build (as a preprocessing step) for the miter circuit under verification helps us to identify these equivalent signals, which are in turn added as two-literal clauses to the existing CNF database.

Consider the circuit shown in Figure 3.2. Its CNF formula is given below:

$$(x \vee p)(y \vee p)(\neg x \vee \neg y \vee \neg p)(x \vee q)(y \vee q)(\neg x \vee \neg y \vee \neg q)$$

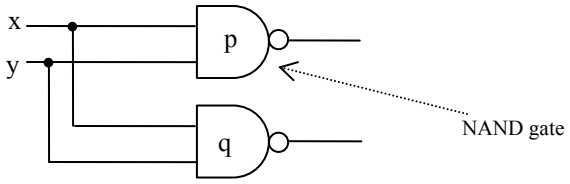


Figure 3.2 Equivalent/Complement Literal Identification

From the CNF formula, we see that the decision $p = 0$ on BCP implies $x = 1$, $y = 1$, and finally $q = 0$. Similarly, the decision $q = 0$ on BCP implies x = *1, y = 1,* and finally $p = 0$. But $p = 1$ implies nothing on q; likewise, $q=1$ implies nothing on p. Hence, we cannot deduce that the two signals p and q are equivalent. However, our implication engine can deduce this relation. We see that *impl* $[p, 0] = \{(p, 0), (x, 1), (y, 1), (q, 0)\}$, where $(p, 0) \rightarrow (q, 0)$ is an indirect implication. By the contrapostive law, $(q, 1) \rightarrow (p, 1)$. Similarly, *impl* $[q, 0]=\{(q, 0), (x, 1),(y, 1), (p, 0)\}$, such that $(q, 0) \rightarrow (p, 0)$ is an indirect implication. Again, using the contrapositive law, $(p, 1) \rightarrow (q, 1)$. Thus, $p \leftrightarrow q$. Therefore, for the two indirect implications, $(p, 0) \rightarrow (q, 0)$ and $(q, 0) \rightarrow (p, 0)$, we add up the clauses $(p \vee \neg q)$ and $(q \vee \neg p)$, respectively. The addition of such two clauses proves the equivalence of two literals p and q. It should be noted that every two-literal clause we add embeds in itself both the indirect implication as well as its contrapostive. Similar to equivalent literals, our approach can also identify complementary signals in the circuit. These relations between intermediate points of the circuit propagate in the forward direction, and help to identify additional relations and implications throughout the circuit.

3.1.3 Identification of Constant/Impossible Nodes

In order to prove the equivalence of two circuits, the corresponding PO's of the two circuits are XOR-ed (i.e., a miter circuit is created), and the XOR outputs are checked if they are at the constant 0 value. In our approach, building the implication

graph for the miter circuit under verification may deduce a few XOR outputs to be constant at logic 0. This happens whenever implications of the following type are obtained:

a. $(Z, 1) \rightarrow (Y, 0)$ and $(Z, 1) \rightarrow (Y, 1)$ or

b. $(Z, 1) \rightarrow (Z, 0)$,

Here Y and Z can be any pair of signals in the miter circuit. The implication of type *(a)* suggests that when Z is set to logic value 1, Y must take on both 0 and 1 as logic values. This is impossible since Y cannot be both 1 and 0 simultaneously. Hence, $Z = 1$ must be impossible, indicating that Z should always be a constant with logic value 0. Similarly, the implication of type *(b)* suggests that $Z = 1$ implies $Z = 0$, i.e. a conflict on itself. This again suggests that $Z = 1$ is impossible and Z has to be a constant with logic value 0. After the implication graph for the miter circuit under verification has been built, all the nodes identified as constants are added as unit literals to the original CNF database. This in turn further prunes the search space of the SAT-solver engine, thereby enhancing its performance.

3.1.4 Significance of Extended Backward Implications

The concept of extended backward implications helps us to learn some very useful, non-trivial two-node implications. When added as two-literal clauses to the original CNF formula, they play a significant role. We will illustrate this by means of the example circuit in Figure 2.1, Chapter 2. The corresponding CNF formula for this circuit has been given earlier. Now, suppose our objective is to satisfy $m = 1$. The first decision that the SAT solver makes is $m = 1$ (given objective), followed by $f = 0$, and then $a = 0$.

The resulting decision tree is shown in Figure 3.3. We see that assigning $a = 0$ results in a conflict. Also, on backtracking $a = 1$ yields a conflict. The SAT-solver again backtracks and sets $f = 1$, and finally the decisions $d = 0$, $b = 0$ make the formula satisfiable.

Now we use our implication engine as a preprocessing step. From extended backward implications, we learned that $f = 0 \rightarrow m = 0$. Applying the contrapostive law, we obtain $m = 1 \rightarrow f = 1$. Hence, we statically insert the clause $(f \vee \neg m)$ in the original CNF formula. Now, if we ask the SAT-solver to satisfy the objective $m = 1, f = 1$ will be implied, and our decision tree will be as shown in Figure 3.4. Adding the two-literal clause results in fewer decisions with no backtracks, and at the same time improves the Boolean Constrain Propagation (BCP).

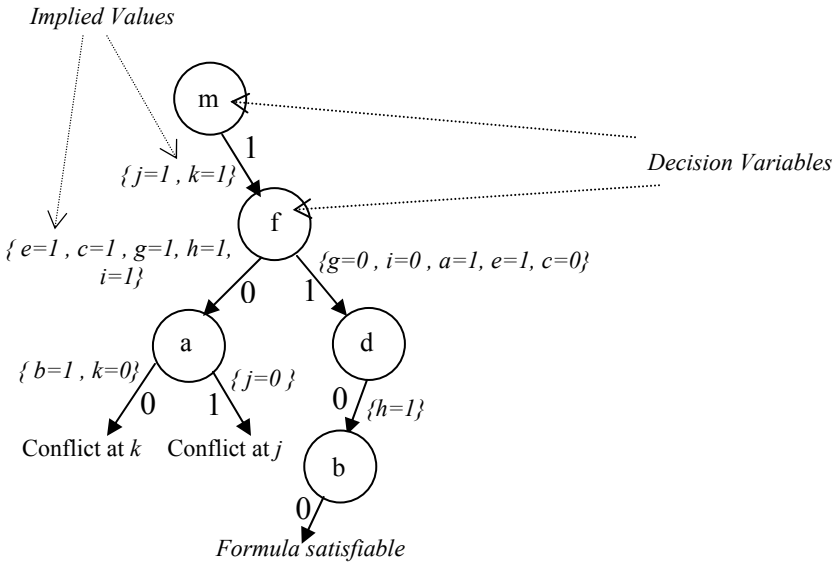


Figure 3.3 Decision Tree without adding any Clauses

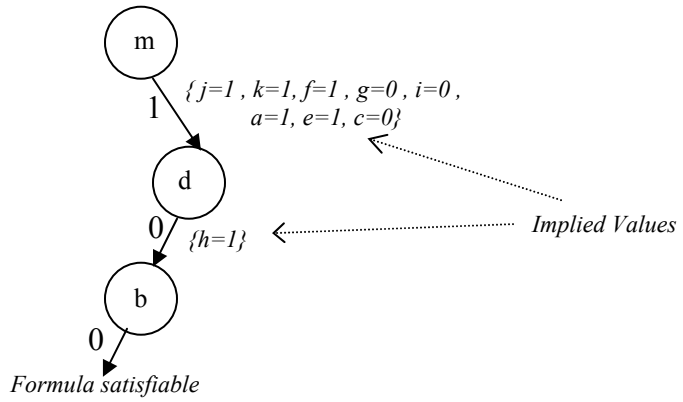


Figure 3.4 Decision Tree after adding the two-literal Clause ($f \vee \neg m$) corresponding to the Extended Backward Implication

3.2 Comparison of our method with Hypre Preprocessor [Bacchus 03]

Our preprocessing engine, when compared with *Hypre* [Bacchus 03], yields much more powerful implications. This was experimentally verified by running *Hypre* [Bacchus 03] on the example circuit of Figure 2.1. It was observed that *Hypre* was not able to deduce the two-literal clause ($f \vee \neg m$). We then ran *Hypre* on another example circuit shown in Figure 3.5. In this case, our preprocessing tool deduced *six* additional non-trivial clauses. On the other hand, *Hypre* deduced only three clauses. All clauses deduced by our method are listed below, in which only half of them (3 clauses) were obtained by *Hypre*:

- $(\neg c \vee g), (f \vee i), (\neg f \vee k)$ $\rightarrow$ *deduced by Hypre as well*
- $(f \vee \neg m), (p \vee \neg k), (p \vee \neg a)$ $\rightarrow$ *deduced only by our preprocessing tool.*

Here, the clause $(f \vee i)$ is obtained by computing indirect implications for node *(f, 0)*, the clauses $(\neg c \vee g)$, $(\neg f \vee k)$ and $(f \vee \neg m)$ are deduced by computing extended backward implications for nodes *(g, 0)*, *(k, 0)* and *(f, 0)*, respectively. And finally the above implication relations help to deduce the non-trivial clauses $(p \vee \neg k)$ and $(p \vee \neg a)$, by

performing extended backward implications on (*p, 0*). This corroborates the fact that our technique is more powerful than *Hypre,* since more implications can be obtained by our method. In section 3.5, we give more experimental results, which further underpin the superiority of our technique.

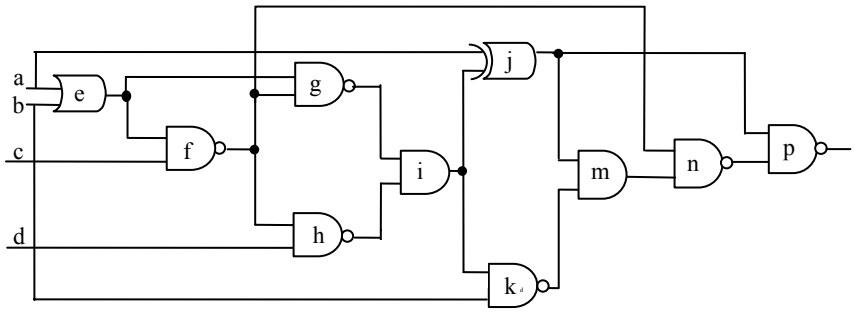


Figure 3.5 Example Combinational Circuit

3.3 Mapping Static Implications on to the CNF formula

In this section, we show how static implications consisting of direct, indirect and extended backward implications can be mapped onto the CNF formula. We also provide a suite of lemmas and theorems to prove that these implications when added as two-literal clauses to the existing CNF database will preserve the accuracy of the CNF formula.

Consider below the CNF formula Φ for the example circuit of Figure 2.1:

$\omega1 = (\neg a \vee e)$, $\omega2 = (\neg b \vee e)$, $\omega3 = (\neg e \vee a \vee b)$, $\omega4 = (f \vee e)$, $\omega5 = (f \vee c)$

$\omega6 = (\neg f \vee \neg e \vee \neg c)$, $\omega7 = (\neg f \vee \neg g)$, $\omega8 = (f \vee g)$, $\omega9 = (\neg f \vee \neg d \vee \neg h)$

$\omega10 = (f \vee h)$, $\omega11 = (d \vee h)$, $\omega12 = (g \vee \neg i)$, $\omega13 = (h \vee \neg i)$, $\omega14 = (\neg g \vee \neg h \vee i)$

$\omega15 = (\neg j \vee a \vee i)$, $\omega16 = (\neg j \vee \neg a \vee \neg i)$, $\omega17 = (j \vee \neg a \vee i)$, $\omega18 = (j \vee a \vee \neg i)$

$\omega19 = (h \vee k)$, $\omega20 = (b \vee k)$, $\omega21 = (\neg h \vee \neg b \vee \neg k)$, $\omega22 = (j \vee \neg m)$, $\omega23 = (k \vee \neg m)$

$\omega24 = (m \vee \neg j \vee \neg k)$

3.3.1 Direct implications in the CNF formula

Direct implications are obtained by *single* application of unit-clause rule to the CNF formula, when each of the variables is set to logic value 1 or 0. For example, when *f* is set to logic value 0, the values implied by application of unit-clause rule to clauses ω4, ω5, ω8, ω10 are (*e, 1*), (*c, 1*), (*g, 1*) and (*h, 1*), respectively. These implications are already embedded in the original CNF formula.

3.3.2 Mapping indirect implications onto the CNF formula

Indirect implications are obtained by *repeated* application of unit-clause rule to the CNF formula (i.e. by performing Boolean Constraint Propagation), when each of the variables is set to logic value 1 or 0. For example, when *f* is set to logic value 0, the values implied by single application of the unit-clause rule are (*e, 1*), (*c, 1*), (*g, 1*) and (*h, 1*) and applying unit-clause rule again yields (*i, 1*) from ω14. We thereby add up the clause $(f \vee i)$ corresponding to this indirect implication.

We thus give a Lemma and a Theorem stating that the clauses added through indirect implications are implicates of the CNF formula, and thus preserve its correctness.

Lemma 1. Given a CNF formula Φ, if $(y, 1) \in \text{BCP}(x, 1)$, then the clause $(\neg x \vee y)$ is an implicate of Φ.

Proof: We prove this by contradiction**.** Assume that the clause $(\neg x \vee y)$ is not an implicate of Φ. Then, two cases can arise:

1. (x, *1*) $\in$ BCP (y, *0*). This is not possible since we are given that (y, *1*) $\in$ BCP (x, *1*), which in turn will yield a conflict on y.
2. (y, *0*) $\in$ BCP (x, *1*). This contradicts with the given condition that (y, *1*) $\in$ BCP (x, *1*).

Hence, the clause ($\neg x \vee y$) is an implicate of Φ and preserves the accuracy of the CNF formula.

Theorem 1. Given a CNF formula Φ, if (y_i, *1*) $\in$ BCP (x, *1*), $i = 1, 2, \ldots n$, then the clauses ($\neg x \vee y_i$) are implicates of Φ.

Proof: The theorem directly follows *Lemma 1*. If a single clause ($\neg x \vee y$) is an implicate of Φ under the above condition, then all the clauses ($x \vee y_i$) where $i = 1, 2, \ldots n$ are implicates of Φ.

Corollary 1: The set of clauses obtained by Theorem 1 contains all the clauses obtained using indirect implications.

Indirect implications of gate G set to value v are obtained by performing logic simulation with direct implications of (G, v) plugged onto the circuit. This is similar to doing BCP (i.e. repeated application of unit-clause rule to the CNF formula) when the CNF variable G is set to value v. Thus, the above corollary will always hold.

3.3.3 Mapping Extended Backward Implications onto the CNF Formula

Extended Backward implications are obtained by scanning the clauses of the CNF formula, Φ for satisfiability. We explain this with the help of an example. Consider again

the CNF formula Φ, for the example circuit of Figure 2.1. We assign (*f, 0*), perform BCP (*f, 0*) and find that along with other clauses, the clause ω3 is still not satisfied. It can be satisfied by setting (*a, 1*) or (*b, 1*). Hence, the implied values common to BCP(*a, 1*) and BCP(*b, 1*) will be the inferred assignments under (*f, 0*). In other words, the set {BCP(*a=1* and *f=0*) ∩ BCP(*b=1* and *f=0*)} will yield the inferred assignments. In this case (*m, 0*) is the common assignment and hence we can add up the clause ($f \vee \neg m$) to the existing CNF database.

We now give Lemmas and Theorem stating that the clauses added through extended backward implications are implicates of the CNF formula, and thus preserve its correctness.

Lemma 2: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if $(y, 1) \in [\cap^{n}_{i=1}(\text{BCP}(l_k, 1))]$, then $(y, 1)$ will be a necessary assignment of Φ.

For the original CNF formula Φ to be satisfied, every clause $\omega \in \Phi$ needs to be satisfied. Clause ω can be satisfied by setting any of its literals to logic 1. Therefore, any common assignment obtained by setting each of the literals in ω to logic 1, will be a necessary assignment.

Proof: We are given that the following:

- Clause ω has *n* literals, i.e. $\omega = (l_1 \vee l_2 \vee \ldots. l_n)$, and
- BCP (l_1, *1*) implies (*y, 1*), ……(*1*)
- BCP (l_2, *1*) implies (*y, 1*),…….(*2*)
- ……..
- BCP (l_n, *1*) implies (*y, 1*). ……(*n*)

We prove this Lemma by contradiction.

Suppose that ($y, 1$) is not a necessary assignment. In other words, there exists a satisfying assignment to the CNF formula with ($y, 0$). However, using equations (1) to (n) by the contrapositive law we obtain $(y, 0) \rightarrow (l_1, 0)$, $(y, 0) \rightarrow (l_2, 0)$, …. $(y, 0) \rightarrow (l_n, 0)$. Since ($y, 0$) implies each of the literals $l_1, l_2, \ldots, l_n$ to logic 0, the clause ω would evaluate to 0, causing the CNF formula to become unsatisfiable. Hence, our assumption is wrong and the assignment ($y, 0$) is not possible. Therefore, ($y, 1$) is a necessary assignment of Φ.

Lemma 3: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if under the assignment ($x, 0$), the literals $l_1, l_2 \ldots. l_j$ ($j < n$) are implied to 0, and if $(y, 1) \in [\cap^{n}_{k=j+1} \text{BCP}\ (l_k = 1 \text{ and } x = 0)]$, then $(x \vee y)$ will be an implicate of Φ.

This Lemma is an extension of Lemma 2 and states that if the current assignment ($x, 0$) implies the literals $l_1, l_2 \ldots. l_j$ ($j < n$) of ω to logic 0, then the common assignment ($y, 1$) obtained by setting each of the remaining literals of ω to 1, together with the current assignment ($x, 0$) will result in an implicate $(x \vee y)$ of Φ.

Proof: We know that for the original CNF formula Φ to be satisfied, every clause $\omega \in \Phi$ needs to be satisfied. If the current assignment ($x, 0$) causes the literals $l_1, l_2 \ldots. l_j$ ($j < n$) of ω to evaluate to 0, the clause ω can only be satisfied if any of its remaining literals evaluate to true (logic 1). Then, the common assignment obtained by setting each of the remaining literals to logic 1 will be a necessary assignment under the condition ($x, 0$). In other words $(x, 0) \rightarrow (y, 1)$ in Φ, or $(x \vee y)$ is an implicate of Φ.

We continue the proof by contradiction. It is given that the assignment ($x, 0$) results in the following:

- $l_1, l_2 \ldots\ldots, l_j$ are implied to 0, and

- BCP (l_{j+1}=*1* and x =*0*) implies (*y, 1*) ……(*1*)
- BCP (l_{j+2} =*1* and x ,*0*) implies (*y, 1*)…….(*2*)
- ……..
- BCP (l_n=*1* and x =*0*) implies (*y, 1*)……..(*n-j*)

Now, suppose ($x \vee y$) is not an implicate of Φ. Then, two cases can arise:

1. (*y, 0*) ∈ BCP (*x, 0*). But by contrapositive law, using equations (*1*) to (*n - j*), (*y, 0*) will imply the following:
    - $(y \vee x \vee \neg l_{j+1})$
    - $(y \vee x \vee \neg l_{j+2})$
    - ……
    - $(y \vee x \vee \neg l_n)$

    However, under the condition (*x, 0*), the above constraints will cause the literals l_{j+1}, $l_{j+2,}$ … l_n to be implied to logic 0, since (*y, 0*) ∈ BCP (*x, 0*). Also, the assignment (*x, 0*) already implies l_1, l_2 …. l_j to logic 0 (given). Thus, the clause ω would evaluate to 0 and the CNF formula will become unsatisfiable
2. (*x, 0*) ∈ BCP (*y, 0*). But by the contrapositive law, using equations (*1*) to (*n - j*), (*y, 0*) will imply the following:
    - $(y \vee x \vee \neg l_{j+1})$
    - $(y \vee x \vee \neg l_{j+2})$
    - ……
    - $(y \vee x \vee \neg l_n)$

    However, under the condition (*y, 0*), the above constraints will cause the literals l_{j+1}, $l_{j+2,}$ … l_n to be implied to logic 0, since (*x, 0*) ∈ BCP (*y, 0*). Also, (*x, 0*) implies the

literals l_1, l_2 …. l_j to 0 (given). Therefore, the clause ω would evaluate to 0 and the CNF formula will become unsatisfiable.

Since the above two cases are not possible, $(x \vee y)$ is an implicate of Φ.

Theorem 2: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if under the assignment $(x, 0)$, the literals l_1, l_2 …. l_j $(j < n)$ are implied to 0, then for every $(y_i, 1) \in [\cap^{n}_{k=j+1} \text{BCP}\ (l_k = 1 \text{ and } x = 0)]$, $i = 1, 2, ...m$, $(x \vee y_i)$ is an implicate of Φ.

Proof: The theorem directly follows *Lemma 3*. If a single clause $(x \vee y)$ is an implicate of Φ under the above condition, then all the clauses $(x \vee y_i)$ where $i = 1, 2, ...m$ are implicates of Φ.

Corollary 2: The set of clauses obtained by Theorem 2 are a superset of all the clauses obtained through extended backward implications

The clauses obtained through extended backward implications are actually a subset of the clauses obtained through Theorem 2. The reason is that we work on the circuit netlist, and while computing extended backward implications we consider only the *unjustified output specified gates* in the implication list of the *target gate*. Hence, not all the clauses $\omega \in \Phi$ are checked for satisfiability under the current assignment. For example, under the assignment $(f, 0)$, Theorem 2 will cause the clauses ω3, ω15, ω16, ω17, ω18 and ω21 to be checked for satisfiability, whereas extended backward implications only checks the clause ω3 for satisfiability. This reduces the computation complexity, although at a cost of loosing some highly non-trivial implications which could otherwise be obtained by a complete implementation of Theorem 2. In Chapter 6,

we provide results obtained after the complete implementation of Theorem 2 as well some new theorems.

3.4 The Algorithm

The flow of our algorithm is described below.

Algorithm:

Step 1. Generate the CNF formula for the miter circuit under verification.

Step 2. Compute the direct and indirect implications for each of the nodes in a *levelized* fashion, from the primary inputs towards the primary outputs.

Step 3. (a) Convert the indirect implications obtained in *Step 2* into two-literal clauses. (b) Append these new clauses to the CNF database.
(c) Add the nodes identified as constants, as *unit* clauses.

Step 4. If more than *n %* of the mitered XOR outputs have been identified as constant *0's*, go to *Step* 7, else go to *Step 5*.

Step 5. For each gate N, compute its extended backward implications.

Step 6. Convert the extended backward implications obtained in *Step 5* into two-literal clauses, and append them to the existing CNF formula.

Step 7. Give the modified CNF formula to the SAT-solver for processing.

Step 8. Stop.

3.5 Experimental Results

The algorithm presented in Section 3.4 was implemented in C++ in a preprocessing engine called *IMP2C* (Implications to Clauses). *IMP2C* builds the

Implication Graph for the miter circuit under verification, and formulates the two-literal clauses corresponding to indirect and extended backward implications learnt. The experiments were run on a Pentium 4, 1.8-GHz machine, with 512 MB of RAM and Mandrake Linux 7.2 as the operating system. The efficacy of our technique is corroborated by using the *large* and *difficult* ISCAS'85 [Brglez 85] benchmark circuits, the ISCAS'89 [Brglez 89] full scan circuits, the ITC'99 [Corno 00] full scan circuits and some cascaded ITC'99 benchmarks. Two different types of miter circuits were verified for equivalence: *circuit_equiv* represents an equivalence checking circuit model where two identical copies of the same circuit are mitered, *circuit_opt* represents mitering of the original copy of the circuit and its optimized version (obtained using *Synopsys*). For both miter circuit types, we OR all the mitered outputs, and ask the SAT solver to satisfy *OR gate* output to logic *one.* We used two different state-of-the-art SAT-solvers, namely, BerkMin561 [Goldberg 02b] and Siege_v4 [Ryan 03] to check the satisfiability of each of the Combinational Equivalence Checking (CEC) instances. Experiments were also run with ZChaff 2001.2.17 [ZhangL 01], but the results have not been reported since for most of the instances ZChaff [ZhangL 01] was found to be 2-10 times slower than BerkMin [Goldberg 02b] and Siege [Ryan 03].

3.5.1 Comparison of SAT-solver Performance without and with IMP2C Preprocessing

In Table 3.1, for each miter circuit, we report the execution time taken by our preprocessing engine IMP2C, the time taken by the SAT-solver alone, and the time taken by IMP2C + SAT-solver together. We also report the speedup ratio of IMP2C +

SAT-solver over SAT-solver alone. The results are reported by arbitrarily choosing *n = 25%* in *Step 4* of the implementation algorithm described in Section 3.4. However, it should be noted that our preprocessor is tuned to handle any threshold given at run time.

Table 3.1 Results with *SAT-solver* alone and (*IMP2C + SAT-solver*)

Miter Circuit	IMP2C (secs)	BerkMin (secs)	IMP2C + BerkMin (secs)	Speed-up	Siege (secs)	IMP2C + Siege (secs)	Speed-up
c499_equiv	0.22	0.28	0.23	1.21	0.3	0.23	1.30
c880_equiv	0.03	0.29	0.03	9.66	0.49	0.04	12.25
c1355_equiv	0.06	0.32	0.06	5.33	1.57	0.07	22.42
c1908_equiv	0.07	0.47	0.07	6.71	2.82	0.08	**35.25**
c2670_equiv	0.29	0.81	0.42	1.92	1.18	0.30	3.93
c3540_equiv	0.94	14.54	0.96	15.14	22.21	0.97	**22.89**
c5315_equiv	0.68	8.76	0.88	9.95	12.04	0.69	17.44
c7552_equiv	1.71	102.31	2.00	51.15	34.52	1.76	19.61
c880_opt	0.14	0.47	0.14	3.35	0.84	0.15	5.6
c1355_opt	0.70	0.27	0.70	0.38	1.26	0.71	1.77
c1908_opt	3.26	0.51	3.26	0.15	2.32	3.27	0.71
c3540_opt	0.82	12.25	0.84	14.58	30.34	1.41	21.51
c5315_opt	16.24	11.77	16.24	0.72	16.23	16.25	0.99
c7552_opt	30.47	168.37	30.48	5.52	39.61	30.48	1.29
s38417_fs_equiv	62.77	272.54	101.14	2.69	336.02	88.6	3.79
s38584.1_fs_equiv	240.02	225.82	291.97	0.77	131.76	267.47	0.49
s35932_fs_equiv	66.28	139.59	87.66	1.59	97.27	81.58	1.19
b14_equiv	26.05	25,168.2	30.42	**827.35**	417.13	27.67	15.07
b14_1_equiv	14.50	8,707.58	17.03	511.31	284.2	15.77	18.02
b15_optim_equiv	57.72	145.55	76.41	1.90	73.9	69.38	1.06
b17_optim_equiv	245.02	3,055.96	330.19	9.25	458.04	316.08	1.44
b18_optim_equiv	2,132.5	>150,000	2,659.77	>56.39	5,780.29	2,557.29	2.26
b20_1_equiv	27.88	13,582.4	36.30	374.17	396.96	36.73	10.80
b21_1_equiv	29.61	14,074.85	38.73	363.40	427.63	37.37	11.44
b22_1_optim_equiv	43.11	16,311.23	57.87	281.85	507.00	61.33	8.26
cascade_1*	380.72	82,456.2	480.34	171.66	2,785.3	440.26	6.32
cascade_2	124.32	74,083.7	158.45	467.55	1,892.65	140.62	13.45
cascade_3	2,584.2	>150,000	3,389.46	>44.25	6,217.78	2,947.12	2.10
cascade_4	428.67	3,822.72	540.52	7.07	8,842.92	470.23	18.8
cascade_5	102.23	60,451.7	145.87	414.42	1654.41	129.65	12.76

*cascade_1 = b17_optim_b14_equiv, cascade_2 = b14_b22_1_optim_equiv, cascade_3 = b18_optim_b15_optim_equiv, cascade_4=b17_optim_b15_optim_equiv, cascade_5 = b20_1_b21_1_equiv

From Table 3.1, we see that considerable speedup is achieved for almost all the instances. In some cases, once the implication relations are computed, the SAT-solver

can determine the formula to be unsatisfiable almost immediately. For instance, in the miter circuits *c7552_equiv* and *c7552_opt*, without any added clauses, BerkMin spent *102.31 seconds* and *168.37 seconds*, respectively. When we augment the CNF formula with the global implication relations (derived by IMP2C), the complexity of the CNF instance is notably reduced, with IMP2C + BerkMin taking (*1.71 + 0.29*) *2.00 seconds* and (*3.47 + 0.01*) *3.48 seconds*, respectively. Note that the SAT-solver BerkMin takes only a fraction of a second. For the instance *b18_optim_equiv,* BerkMin alone could not finish even after *150,000 seconds,* but after IMP2C clauses are added the instance is solved in (*527.27 +2,132.5*) *2,659.77 seconds*; the time taken by BerkMin being *527.27 seconds* and the time taken by IMP2C being *2,132.5 seconds*.

Similarly, the other SAT-solver Siege also yields significant speedup with our preprocessing technique. It is observed that Siege generally outperforms BerkMin for almost all the instances and hence the speedups with Siege are somewhat smaller than with BerkMin. For some of the relatively easier CEC instances (*e.g. c5315_opt, s38584.1_fs_equiv*), the preprocessing due to indirect and extended backward implications was a bit of overhead, and thus not much speedup was achieved. However, it should be noted that after our preprocessing has been applied, the time taken by the SAT-solver *alone* reduces significantly for all the instances. This suggests that the clauses added are extremely meaningful and cause considerable search space pruning, reducing the SAT instance complexity immensely. Overall, the results for IMP2C + SAT-solver are very encouraging, with maximum speedup for IMP2C + BerkMin being *827.35X,* and for IMP2C + Siege being *22.89X.* Since considerable speedup is achieved with each of the SAT-solvers, our approach is orthogonal to the SAT-solver used.

The ISCAS'85 benchmark *c6288* is a 16-bit multiplier circuit and its corresponding miter instances are known to be very difficult for SAT-solvers. Hence, we have treated *c6288_equiv* and *c6288_opt* instances in a separate Table. Table 3.2 shows the performance of each of the SAT-solvers for these instances without and with our preprocessing technique. The results show that the SAT-solver BerkMin (alone) could not finish even after *7200 seconds.* The Siege SAT-solver could solve the *c6288_equiv* and *c6288_opt* miters in *4852.3 seconds* and *5214.5 seconds*, respectively. However, with our preprocessing technique (IMP2C), the two instances were quickly solved in less than one-tenth of the second, the preprocessing time being *0.35 seconds* and *3.88 seconds* for *c6288_equiv* and *c6288_opt*, respectively.

Table 3.2 Results for *c6288* with *SAT-solver* alone and *IMP2C+SAT-solver*

Miter Circuit	IMP2C (secs)	BerkMin (secs)	IMP2C + BerkMin (secs)	Speed-up	Siege (secs)	IMP2C + Siege (secs)	Speed-up
c6288_equiv	0.35	>7,200	0.35	> 20,571.4	4,852.3	0.36	13,478.6
c6288_opt	3.88	>7,200	3.89	> 1,850.8	5,214.5	3.90	1,337.05

In Table 3.3 we give the number of clauses in the original CNF, the time taken by our preprocessing technique (IMP2C), the number of clauses added using IMP2C, and finally the ratio of added clauses to original clauses. We see from Table 3.3 that as the size of the circuit (# original clauses) increases, the time for IMP2C increases in proportion, since many nodes need to be processed. Also, some circuit structures are such that there are a lot of implication relations among the nodes and hence IMP2C takes a long time. One such case is *b15_equiv* for which IMP2C deduced more than twice the number of clauses that were in the original CNF formula. It must be noted that even though many clauses were added, we achieve noteworthy speedup for almost all cases,

suggesting that the clauses deduced were extremely helpful in pruning the SAT-solver search space. Overall, the ratio of added clauses to original clauses varied from *0.29* for *s38417_equiv* to *2.37* for *b15_equiv,* with the mean being around *0.95.*

Table 3.3. Number of Original and Added Clauses for different CEC instances

Miter Circuit	Original #Clauses	IMP2C (secs)	Added #Clauses (IMP2C)	Added #Clauses/ Original #Clauses
c3540_equiv	9,462	0.94	4,116	0.44
c5315_equiv	15,743	0.68	6,123	0.39
c6288_equiv	14,788	0.35	6,956	0.47
c7552_equiv	20,504	1.71	13,080	0.64
c3540_opt	9,262	0.82	3,780	0.40
c5315_opt	14,151	16.24	7,261	0.51
c6288_opt	14,719	3.88	8,700	0.59
c7552_opt	20,111	30.47	11,800	0.59
s38417_ equiv	127,580	62.77	38,029	0.29
s38584.1_ equiv	123,052	240.02	51,894	0.42
s35932_fs_equiv	111,200	66.28	39,977	0.35
b14_equiv	60,661	26.05	75,980	1.25
b14_1_equiv	42,203	14.50	45,968	1.09
b15_optim_equiv	51,329	57.72	121,928	2.37
b17_optim_equiv	165,189	245.02	361,882	2.19
b18_optim_equiv	486,717	2,132.5	866,832	1.78
b20_1_equiv	87,582	27.88	73,379	0.83
b21_1_equiv	87,760	29.61	80,483	0.93
b22_1_optim_equiv	103,173	43.11	84,789	0.82
cascade_1	226,143	380.72	405,874	1.79
cascade_2	164,043	124.32	220,174	1.34
cascade_3	537,289	2,584.2	1,006,745	1.87
cascade_4	217,054	428.67	462,183	2.12
cascade_5	175,985	102.23	168,143	0.95

3.5.2 Comparison of IMP2C with other Preprocessing Techniques

In Table 3.4, we compare our results with C-SAT-Jnode [Lu 03a], P_EQ + Berkmin [Novikov 03] and Hypre [Bacchus 03] for ISCAS'85 *circuit_equiv* versions. In [Lu 03a], the authors introduced *incremental learn-from-conflict* strategy. Their algorithm divides the problem at hand into unsatisfiable sub-problems and adds the

conflict-induced clauses resulting from solving these sub-problems to the original CNF formula. In [Bacchus 03], the authors utilize hyper binary resolution and equality reduction to simplify the CNF formula. Their tool Hypre can either prove the unsatisfiability of the given CNF formula or yield a simplified CNF formula with fewer variables and clauses. The *circuit_equiv* versions in Table 3.4 were all proved unsatisfiable by Hypre. According to Table 3.4, our results are mostly on the same order of computation effort, and in a few cases better than [Lu 03a, Bacchus 03]. In [Novikov 03], the author gave a theoretical framework for deducing multi-literal relationships. However, a restricted version of the technique was implemented, which deduced only the unit and equivalent literals. In our approach, in addition to deducing unit and equivalent literals we also deduce non-trivial implication relationships as well. These relationships when added to the CNF database are very helpful in reducing the SAT instance complexity as has been shown in the experimental results.

Table 3.4 Comparison of IMP2C with [Lu 03a], [Novikov 03] and [Bacchus 03] for ISCAS' 85 *circuit_equiv*

Miter Circuit	C-SAT-Jnode [Lu 03a] (secs)	P_EQ + Berkmin [Novikov 03][@] (secs)	Hypre [Bacchus 03] (secs)	IMP2C + BerkMin (secs)	IMP2C + Siege (secs)
c1355_equiv	0.07	0.05	0.15	0.06	0.07
c1908_equiv	0.11	0.27	0.14	0.07	0.08
c2670_equiv	0.13	0.17	0.13	0.42	0.30
c3540_equiv	1.21	0.83	0.86	0.96	0.97
c5315_equiv	0.28	0.61	0.68	0.88	0.69
c6288_equiv	4.14	0.17	0.98	0.35	0.36
c7552_equiv	1.62	0.87	1.48	2.00	1.76

[@] Expts. were run on P-3, 700MHz with 640Mb RAM [Novikov 03]

3.5.3 Comparison of IMP2C with Hypre [Bacchus 03]

We also performed another set of experiments to show that the clauses obtained through our preprocessing technique are more powerful and non-trivial than those

obtained through Hypre [Bacchus 03]. The results substantiating this are shown in Table 3.5. The *circuit_opt* CNF instances shown here could not be proved unsatisfiable by Hypre alone and the resulting simplified CNF formula was given to Siege for processing. The CNF instance *c7552_1_opt* used here is much more optimized than *c7552_opt* used in Table 3.1; *c7552_opt* was proved unsatisfiable by Hypre alone and did not yield any simplified formula. For the *circuit_equiv* versions in Table 3.5, Hypre did not yield any

Table 3.5 Comparison of IMP2C with Hypre [Bacchus 03]

Miter Circuit	Hypre + Siege (secs)	IMP2C + Siege (secs)	Speed up (col 1 / col 3)	IMP2C+ Hypre+ Siege (secs)	Speed up (col 1 / col 5)
c3540_opt	1.58	1.16	1.36	1.44	1.09
c7552_1_opt	8.68	12.24	0.70	7.24	**1.20**
s38417_fs_equiv	SF*	88.6	----------	SF	----------
s38584.1_fs_equiv	SF	267.47	---------	SF	----------
s35932_fs_equiv	SF	81.58	----------	SF	----------
b14_equiv	74.2	27.67	2.68	30.07	2.46
b14_1_equiv	24.5	15.77	1.55	15.97	1.53
b15_optim_equiv	400.12	69.38	**5.76**	254.29	1.57
b17_optim_equiv	SF	316.08	----------	SF	---------
b18_optim_equiv	SF	2557.29	----------	SF	----------
b21_1_equiv	71.29	37.37	1.90	34.57	2.06
b20_1_equiv	65.33	36.73	1.77	33.60	1.94
b22_1_optim_equiv	105.81	61.33	1.72	46.36	2.28
cascade_1	SF	440.26	----------	SF	----------
cascade_2	186.23	140.62	1.32	134.54	1.38
cascade_3	SF	2,947.12	----------	SF	----------
cascade_4	SF	470.23	----------	SF	----------
cascade_5	265.67	129.65	2.04	117.41	2.26

* SF – Segmentation Fault

simplified formula and proved the unsatisfiability immediately. Therefore, for these instances in columns 2 and 5 we take the Siege time to be *0.0 seconds*. For each of the circuits we give the time taken by Hyper + Siege together, followed by total the time taken by IMP2C + Siege. In column 4 we give the speedup of IMP2C + Siege over Hyper

+ Siege. It was observed that when the augmented CNF formula (with IMP2C clauses) was given to Hypre for preprocessing and the resulting simplified formula to Siege, the time to prove unsatisfiability further reduced. The results for this are given in column 5. In column 6 we give the speedup of (IMP2C + Hypre + Siege) over (Hypre + Siege). For a few of the larger instances *s38417_fs_equiv, s38584.1_fs_equiv*, *s35932_fs_equiv, b18_optim_equiv, b17_optim_equiv, cascade_1, cascade_3* and *cascade_5,* Hypre gave segmentation fault since it has a limit on the number of literals it can handle in a clause (maximum clause length allowed being approximately 1000). The results for these instances have therefore not been reported with Hypre.

We see from Table 3.5 that for most of the instances our technique is more superior than Hypre. As is evident from column 4, we consistently get a speed up of close to *2X* with maximum speedup being *5.76X* for the instance *b15_optim_equiv.* It has been shown earlier by means of examples (see section 3.2) that the non-trivial clauses obtained through our approach cannot be obtained through Hypre. We achieve speedups ranging from *1.36X* to *5.76X*. For example, with *b14_equiv* Hypre + Siege spent *74.2 seconds* whereas IMP2C + Siege spent *27.67 seconds* to prove the unsatisfiability, yielding a speedup of *2.68X*. For a total of 6 cases, the speedup in column 6 is slightly greater than in column 4; the reason is that in our approach (IMP2C + Siege), we just augment the original CNF formula with non-trivial two-literal clauses, but do not involve in any equality reduction as is done in Hypre (see Chapter 1, Section 1.1). On the other hand, the CNF formula in column 5 after preprocessing with IMP2C undergoes equality reduction by Hypre, thereby yielding a much simplified and smaller CNF instance. As a result, slightly better execution times are obtained in column 5 than in column 3. One prominent

instance where IMP2C + Siege outperforms IMP2C + Hypre + Siege is *b15_optim_equiv* where IMP2C + Siege took just (*57.72 + 11.66) 69.38 seconds*, where as IMP2C + Hypre + Siege spent (*57.72+ 196.78 + 0.0*) *254.29 seconds* for preprocessing. Here the preprocessing due to Hypre was an overhead and did not help in reducing the overall execution time.

CHAPTER 4

BOOSTING SAT-BASED BOUNDED MODEL CHECKING (BMC) USING SEQUENTIAL IMPLICATIONS

In this chapter we show how sequential logic implications can be used to improve the performance of SAT-based Bounded Model Checking. We present a novel technique for enhancing SAT-based Bounded Model Checking by inducing powerful sequential signal correlations (crossing time-frame boundaries) into the original CNF formula of the unrolled circuit. A quick preprocessing on the circuit-under-verification, builds a large set of direct and indirect sequential implications. The non-trivial implications (spanning multiple time-frames) are converted into two-literal clauses. Also, since these implications are globally true relative to the time frames in which they span, they are quickly replicated throughout the unrolled sequential circuit as per their edge weights, and appended to the existing CNF database in clause form. The added clauses induce global signal correlations among the CNF variables of different time frames. Thus, when a SAT-solver tries to solve this modified CNF instance (augmented with additional meaningful clauses), it experiences a reduction in the number of decisions due to improved Boolean Constraint Propagation and also a reduction in the number of backtracks. This preprocessing results in efficiently constraining the search space, and hence the SAT-solver can determine the satisfiability/unsatisfiability of the CNF instance much more quickly compared to the conventional approach. As per the author's knowledge, this is the first approach of its kind to use signal relationships spanning time-frame boundaries to

enhance the performance of SAT-based BMC. This is unlike the previous work [Gupta 03], where only relationships among signals in the combinational portion of the circuit are learnt. Moreover, our sequential relations are learned *without* unrolling the circuit; thus, the static learning is fast.

Experimental Results for checking difficult instances of random safety properties on ISCAS'89 benchmark circuits show that more than an order of magnitude speedup can be achieved over the conventional approach.

4.1 Application of Sequential Implications to SAT-based BMC

The complexity of the BMC instance formulated as a SAT problem depends on the property to be verified, as well as the underlying SAT solver. In order to speed up the search, relations within the circuit can be very useful because they can help to constrain the search space. Global relations (across time frames) can be extremely useful. We call these global relations as sequential logic implications and partly described them in Chapter 2.

To better explain how sequential implications can be used to improve SAT-based BMC, consider the example circuit of Figure 4.1.

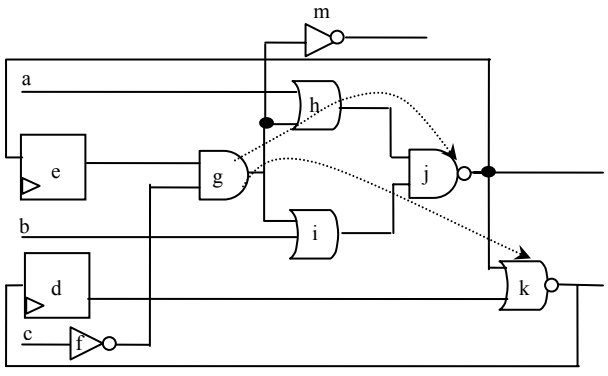


Figure 4.1 Example Sequential Circuit

As an illustration, let us determine the logic implications of gate *g* set to value *1*. The direct implications of (*g, 1*) is given by *impl* [*g, 1*] = {(*g, 1*), (*h, 1*), (*i, 1*),(*m, 0*), (*e, 1*), (*f, 1*)}. The indirect implications of a node can be computed by simply logic simulating the transitive closure of its direct implications, time-frame by time-frame, in an event-driven fashion. The transitive closure of *impl* [*g, 1*] is {(*g, 1*), (*h, 1*),(*i, 1*), (*m, 0*), (*e, 1*), (*j, 1, -1*), (*k, 0, -1*), (*d, 0*), (*f, 1*) , (*c, 0*)}. Now we see that (*h, 1*) or (*i, 1*) individually, do not imply anything on gate *j* (in time-frame *0*). However, together they imply (*j, 0*). Thus, (*g, 1*) → (*j, 0*) is an indirect implication (shown by dotted arrow in Figure 4.1). Now, (*j, 0*) together with (*d, 0*) implies (*k, 1*) in time-frame *0*. Therefore, (*g, 1*) → (*k, 1*) is also an indirect implication (shown by dotted arrow in Figure 4.1). It should be noted that (*k, 1*), an indirect implication of (*g, 1*) is obtained by making use of implications of time-frame *-1*, since (*j, 1, -1*) → (*k, 0, -1*) → (*d, 0*), and hence is a non-trivial implication. These resulting implications are added to the implication graph of the circuit along with their corresponding contrapositive implications. Thus, *impl* [*g, 1*] = {(*g, 1*), (*h, 1*), (*i, 1*),(*m, 0*), (*e, 1*), (*j, 1, -1*), (*k, 0, -1*), (*d, 0*), (*f, 1*), (*c, 0*), (*j, 0*), (*k, 1*),(*d, 1, 1*),(*k, 0, 1*), (*d ,0 ,2*)}. Note that the indirect implications can also cross time frame boundaries. The reader is referred to [Zhao 01] for an in-depth discussion on sequential implications.

4.1.1 Constrained Search Space and Enhanced Boolean Constraint Propagation

In this sub-section, using the above example circuit we show how the sequential logic implications when added as binary clauses to the existing CNF database, help to

improve the BCP and constrain the search space, thereby improving the SAT-solver performance.

A two time-frame unrolled circuit, corresponding to example circuit of Figure 4.1 is shown in Figure 4.2.

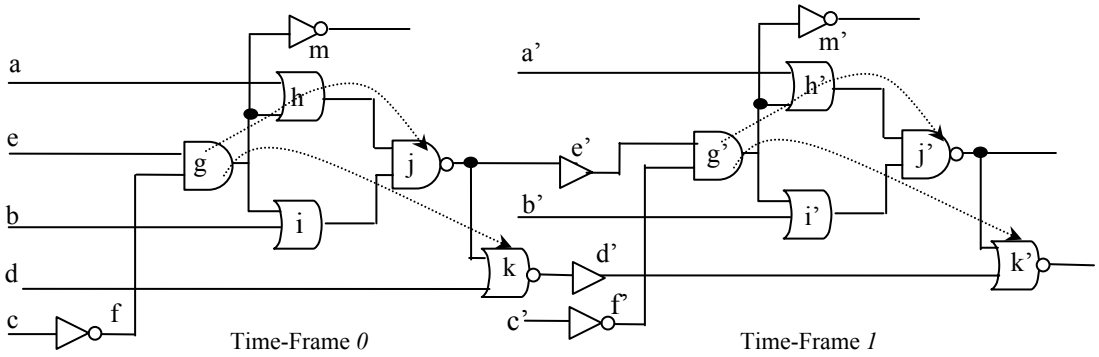


Figure 4.2 Two time-frame unrolled circuit, corresponding to the sequential circuit in Figure 4.1

The partial CNF formula for the unrolled circuit of Figure 4.2 is as follows:

$$(h \vee j)(i \vee j)(\neg h \vee \neg i \vee \neg j)(\neg j \vee \neg k)(\neg d \vee \neg k)(j \vee d \vee k)(j \vee \neg e')$$
$$(\neg j \vee e')(c' \vee f')(\neg c' \vee \neg f')(e' \vee \neg g')(f' \vee \neg g')(\neg e' \vee \neg f' \vee g')$$
$$(\neg a' \vee h')(\neg g' \vee h')\ (a' \vee g' \vee \neg h')(\neg b' \vee i')(\neg g' \vee i')$$
$$(b' \vee g' \vee \neg i')(m' \vee g')(\neg m' \vee \neg g')(h' \vee j')(i' \vee j')(\neg h' \vee \neg i' \vee \neg j')$$
$$(\neg j' \vee \neg k')(\neg d' \vee \neg k')(j' \vee d' \vee k')(k \vee \neg d')(\neg k \vee d')$$

Now let us make the decision $k'=0$. We see that making the decision $k'=0$ satisfies a total of *two* clauses, $(\neg j' \vee \neg k')$ and $(\neg d' \vee \neg k')$. It does not yield any unit clauses. Considering the indirect implications of the corresponding sequential circuit (*Figure 4.1*), we know that $(g, 1) \rightarrow (j, 0)$. Since this implication is globally true in each of the time frames, it can be replicated as $(g', 1) \rightarrow (j', 0)$ (shown as dotted arrow in Figure 4.2). And from contrapositive law, we get $(j', 1) \rightarrow (g', 0)$. Therefore, the two-literal clauses for

time frames *0* and *1* corresponding to the implication $(g, 1) \rightarrow (j, 0)$ are $(\neg g \vee \neg j)$ and $(\neg g' \vee \neg j')$. Also, $(g, 1) \rightarrow (k, 1)$ is an indirect sequential implication. Hence, after replication, $(g', 1) \rightarrow (k', 1)$ is also an indirect implication (shown as dotted arrow in Figure 4.2) with its contrapositive being $(k', 0) \rightarrow (g', 0)$. The two-literal clauses in this case are $(\neg g \vee k)$ and $(\neg g' \vee k')$. As mentioned in Chapter 3, a two-literal clause embeds in itself both the indirect implication as well as its contrapostive.

Let us add the clause $(\neg g' \vee k')$, corresponding to the implication $(g', 1) \rightarrow (k', 1)$ to the existing CNF database. In this case, making the decision $k'=0$, and doing Boolean Constraint Propagation (BCP) will yield unit clauses, implying $g'=0$ and then $m'=1$. This single clause on addition helps to satisfy a total of *eight* clauses instead of *two*. This is shown in Figure 4.3. It is also evident that without adding any clauses, the decision $k'=0$ followed by $g'=1$ causes a conflict, and hence results in backtracking; with the added clause this *backtracking* is avoided. Note that we can add both $(\neg g \vee k)$ and $(\neg g' \vee k')$ to the existing CNF database, but for the illustration purpose we just added $(\neg g' \vee k')$.

In the above example, we added the clause pertaining to combinational implication, although the implication was non-trivially obtained by making use of implications of time frame *-1*. It should however be noted that the sequential implications can cross time-frame boundaries as well, and in that case the replication is done differently as per their edge weights. This is explained in more detail in section 4.1.2. Addition of a large number of these non-trivial two-literal clauses provides correlation among the CNF variables of different time-frames, which act as constraints and prune the overall search space. Also, since we get a considerable improvement in BCP, the total number of decisions to prove the satisfiability/unsatisfiability of the CNF instance is

reduced accordingly. All this leads to noteworthy gain in the SAT-solver performance. These implications between intermediate points of the circuit propagate in the forward/backward direction, crossing the flip-flop boundaries, and hence help to identify global relations throughout the sequential circuit. We term this process of adding sequential implications as two-literal clauses to CNF database as *Sequential Learning.*

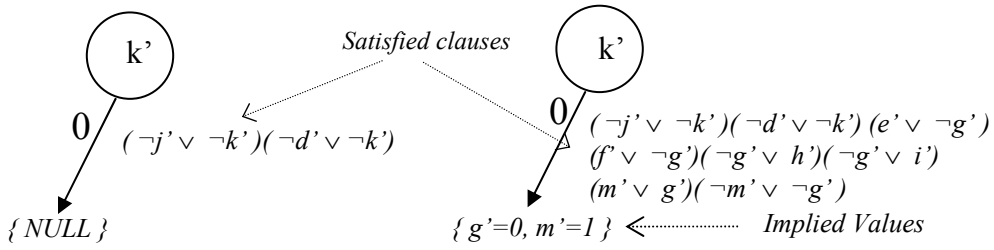


Figure 4.3 Implied values and satisfied clauses in the CNF formula, before and after adding the clause ($\neg g' \vee k'$).

4.1.2 Efficacy of Sequential Implications

Greater sequential implication depth or maximum edge weight allows more *sequential learning*, but requires additional computational effort. Due to reasons cited in section 2.1.4, we make this *sequential implication depth* user specified. In our experiments, a sequential implication depth of *2* (time-frames ranging from -*2* to +*2*) was sufficient to provide a large amount of *learning.* In Figure 4.4, we give the representation of sequential implications in a *5* time-frame unrolled sequential circuit. These implications not only help us to identify relations throughout the combinational portion of the sequential circuit *(of type a → b),* but also the relations spanning multiple time frames (*of type w → x and y → z*) which play a very significant role. We see that *node a → node b* is an implication of depth *0.* This is replicated as *node* $a_0 \rightarrow$ *node* b_0, *node* $a_1 \rightarrow$

node b_1*, node* $a_2 \rightarrow$ *node* b_2 and so on for each of the time- frames, starting from time-frame *0* till time-frame *k-1*. The sequential implications crossing time-frame boundaries are replicated successively from time-frame *0* to time-frame *k-1* as per their edge weights. As seen in Figure 4.4, *node y* $\rightarrow$ *node z* in time-frame *-2* (implication of depth *2*). This is replicated as *node* $y_2 \rightarrow$ *node* z_0*, node* $y_3 \rightarrow$ *node* z_1 and so on in the unrolled circuit; the difference in the time-frames of nodes z_0 and y_2 and also z_1 and y_3 being *-2*. Again, similar replication is done for the implication, *node w* $\rightarrow$ *node x* in time-frame *1*. Note that these sequential implications crossing time-frames (*w* $\rightarrow$ *x and y* $\rightarrow$ *z in the figure*) only need to be computed *once* in our approach, and the subsequent replication is applied automatically. This is entirely different from combinational learning on the unrolled circuit, where each relation crossing time-frame boundary (*each of* $w_i \rightarrow x_i$ *and* $y_i \rightarrow z_i$*)* is regarded as a distinct relation and is learned individually. Such combinational learning on the unrolled circuit would be computationally very expensive, and the resulting performance gain minimal. Also, the computation cost in this case would be proportional to the number of time frames (*k*) the sequential circuit is unrolled into. On the other hand, the sequential learning performed in our case is quite inexpensive, since the computation cost depends just on the sequential implication depth and is independent of the bounded length *k.* These sequential implication relations, especially the ones crossing time-frame boundaries, help to induce meaningful structural information throughout the unrolled sequential circuit, which in turn enhances the SAT-solver performance.

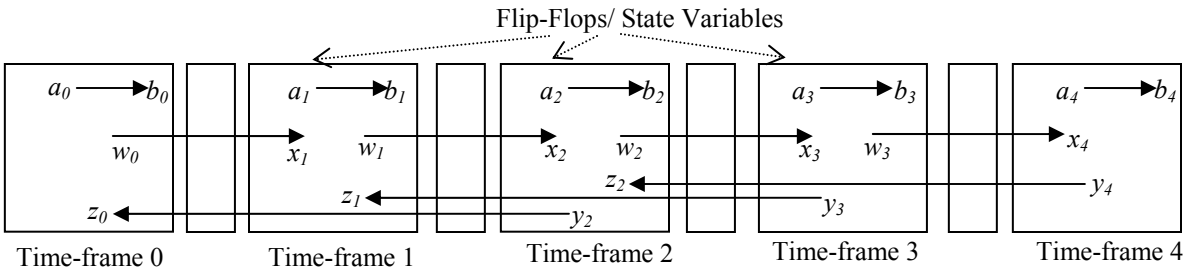


Figure 4.4 Replication of sequential implications in an unrolled circuit.

4.2 The Algorithm

The flow of our *algorithm* is described below:

1. Construct the CNF database for the transformed unrolled sequential circuit, with *monitor circuit* [Boppana 99], as per the property to be verified.
2. Build the Sequential Implication Graph for the original circuit under verification (CUV) for a user defined *sequential implication depth*.
3. Formulate the two-literal clauses corresponding to the indirect implications (*spanning multiple time-frames*) learnt.
4. Replicate these two-literal clauses successively in each of the time-frames as per their edge weights (discussed in section 4.1.2).
5. Appended these clauses to the existing CNF database.
6. Ask the SAT solver to satisfy the *monitor circuit* O/P to logic *1*.

4.3 Experimental Results

The proposed concept was implemented in C++ in a preprocessing engine called *SIMP2C* (Sequential Implications to Clauses). The experiments were run on Pentium-4, 1.8GHz machine, with 512Mb of RAM and Linux as the operating system. Arbitrary

safety properties of the form *EF*(*s*) (where *s* is a complete or partial state) are generated and verified. Liveness properties of the form *EG*(*s*) can also be verified using our method. However, because most *liveness properties* are extremely easy to check for, in the bounded length *k* (since the SAT instance is more constrained), they are omitted in this work. *ZChaff* [ZhangL 01] and *BerkMin* [Goldberg 02b] are used as the SAT solver for all instances. The results for the effectiveness of our approach are shown in Table 4.1. The execution times reported are the average on a set of 10 *random difficult* safety properties for each circuit. The easy properties are quickly solved by the SAT-solver and don't require any preprocessing. It is the difficult instances where our novel technique yields a significant speed-up. These properties include both *satisfiable* and *unsatisfiable* instances. In Table 4.1, for each of the sequential circuits, we give the average execution time taken by our preprocessing engine *SIMP2C;* the number in the parenthesis indicates the *sequential implication depth (seqImp_depth).* We then give the average execution time taken by *ZChaff* and *BerkMin* without any preprocessing and the combined time taken by *SIMP2C + ZChaff* and *SIMP2C + BerkMin*. We also report the speedup obtained by using our preprocessing over the conventional SAT-based approach.

According to Table 4.1, the proposed method (*SIMP2C + SAT-solver*) achieved speedups ranging from *1.24X* for *s4863* to *148.98X* for *s9234.1,* irrespective of the underlying *SAT-solver.* The vast range in speedup is due to the fact that the execution time is both circuit and property dependent. Some properties can be quickly solved by *SAT-solver* alone, whereas some are computationally expensive. For instance, the random properties generated for circuit *s13207.1* were all solved very quickly with *ZChaff* (alone) taking average time of *20.68 secs.* After the non-trivial implication clauses were added

Table 4.1 Average Results for a set of 10 difficult random safety properties on ISCAS'89 benchmark circuits

Circuit Name	#FFs	k (Bound)	SIMP2C (secs.)*	ZChaff (secs.)	SIMP2C + ZChaff (secs.)	Speedup	BerkMin (secs.)	SIMP2C + BerkMin (secs.)	Speedup
s298	14	500	0.14 (2)	24.52	3.21	7.63	21.23	11.39	1.86
s382	21	410	0.24 (4)	64.72	7.93	8.16	75.23	13.67	5.50
s400	21	450	0.22 (2)	58.96	15.11	3.90	38.3	0.89	43.0
s444	21	380	0.15 (2)	33.23	5.38	6.17	34.25	1.04	32.88
s499	22	580	1.29 (1)	27.99	10.58	2.64	81.22	5.14	15.88
s510	6	220	0.35 (2)	357.58	12.12	29.50	1471.3	163.1	9.02
s820	5	410	0.78 (2)	90.03	3.22	27.95	414.6	16.6	24.97
s991	19	320	0.22 (2)	44.11	8.41	5.24	810.24	580.23	1.42
s1423	74	100	0.40 (4)	68.31	32.02	2.13	62.3	8.15	7.71
s1488	6	220	2.67 (2)	56.62	6.58	8.60	714.52	55.72	12.82
s1512	57	190	0.78(2)	654.38	15.52	42.16	440.3	7.7	57.18
s4863	104	98	0.95 (1)	72.26	58.05	1.24	20.18	9.76	2.06
s6669	239	225	1.29 (2)	751.22	58.51	12.83	45.68	18.15	2.51
s9234.1	211	99	8.85 (2)	2867.72	84.27	34.03	1445.2	9.7	**148.98**
s15850.1	534	90	13.2 (1)	42.87	20.69	2.07	349.3	84.6	4.12
s13207.1	638	79	8.42 (1)	20.68	16.41	1.26	23.61	14.99	1.57
s35932	1728	65	23.3 (1)	728.05	52.02	13.99	431.23	75.45	5.71
s38417	1636	85	25.12(1)	380.67	45.34	8.40	585.21	97.32	6.01

*sequential implication depth of (n) in column 5 implies a total of 2n +1 timeframes (ranging from –n to +n including time-frame 0)

using *SIMP2C,* the average time taken by *ZChaff* reduced to *7.99 secs* (not shown), indicating a good amount of search space pruning. However, the total time taken by *SIMP2C + ZChaff* was *16.41 secs* (*8.42 + 7.99*), resulting only in a small speedup of *26%*. On the other hand, for circuit *s9234.1,* the average execution time to solve a set of 10 difficult random safety properties was reduced from *1445.2 secs.(BerkMin alone)* to *9.7 secs (SIMP2C+BerkMin)*, thereby achieving a speedup of *148.98X*.

The time taken by our pre-processing engine *SIMP2C* is very low, ranging from *0.14 seconds* to *25.12 seconds,* thus making our method very attractive; little additional effort is sufficient to reduce the SAT solution complexity.

4.3.1 Effect of increasing the Bounded Length *k*

We observed that in the conventional SAT-based BMC method, increasing the time-frame bound *k* to verify the given safety property causes an *exponential* increase in the execution time. This is shown in Figure 4.5 for the circuits *s510* and *s38417*. The properties being checked here are *Unsatisfiable.* From Figure 4.5, we see that our approach (*SIMP2C* + *ZChaff*) is able to reduce this *exponential* execution time to almost *linear* time.

Table 4.2 Effect of increasing bounded length *k* on SAT-solver performance without and with *SIMP2C*

Circuit	k	ZChaff (secs.)	SIMP2C (secs.)	SIMP2C + ZChaff (secs.)
s510	80	31.33	0.14 (2)	0.58
	120	78.38	0.22 (2)	0.73
	160	128.13	0.29 (2)	7.94
	200	263.69	0.34 (2)	6.71
	240	409.38	0.44 (2)	28.99
s38417	70	151.53	24.01(1)	34.05
	75	242.16	24.37(1)	34.12
	80	333.48	24.67(1)	35.32
	85	427.85	25.43(1)	36.19
	90	538.25	28.17(1)	43.37

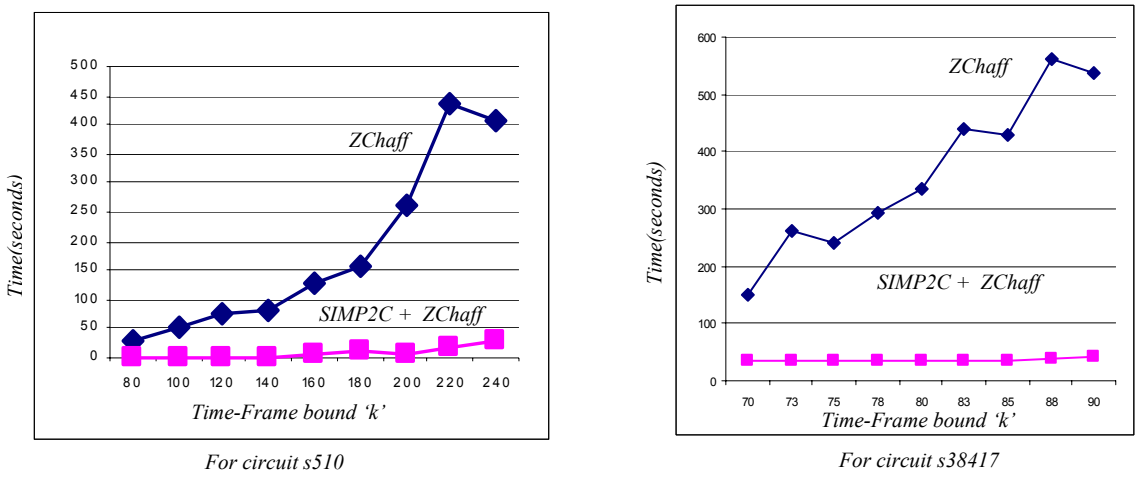


Figure 4.5 Graphical representation of increasing bounded length *k* on SAT-solver performance without and with *SIMP2C*

4.3.2 Effect of increasing the Sequential Implication Depth

Table 4.2 shows the effect of increasing the sequential implication depth on execution time. Greater sequential implication depth allows for greater *sequential learning*, but at an increased cost. From Table 4.2, we see that for circuit *s382,* increasing learning from *seqImp_depth 0* (combinational learning) to *seqImp_depth 4* decreases the average execution time from *26.16* to *13.23 seconds*. For *s35932*, the increase in learning from *seqImp_depth 0* to *seqImp_depth 1* resulted in increased speedup ratio from *4.37*X to *12.25*X. However, increasing sequential learning further to *seqImp_depth* of *2* resulted in *SIMP2C* taking more time. Hence, speedup ratio reduced by a small amount from *12.25*X to *10.77*X. For *s1512* speedup ratio increased from *43.6*X *to 47.96*X as sequential learning was increased from depth *1* to depth *2*.

Table 4.3 Effect of increasing sequential implication depth on SAT-solver Performance without and with SIMP2C

Circuit Name	Property type	#FFs	K (Time Bound)	ZChaff (secs.)	SIMP2C (secs.)	SIMP2C+ ZChaff (secs.)	Speedup
S298	UnSat	14	500	29.21	0.11 (1)	4.78	6.14
					0.14 (2)	3.08	9.48
S382	UnSat	21	410	54.39	0.08 (0)	26.16	2.07
					0.17 (2)	20.13	2.70
					0.27 (4)	13.23	4.11
s1512	UnSat	57	190	857.12	0.6 (1)	19.67	43.60
					0.78 (2)	17.87	47.96
s35932	Sat	1728	65	858.05	9.79 (0)	195.94	4.37
					23.3 (1)	70.02	12.25
					37.05 (2)	79.60	10.77

CHAPTER 5

A NOVEL GLOBAL LEARNING TECHNIQUE

In this chapter, we present a novel global learning technique which strengthens the existing set of static logic implications. We call this new class of implications as *Extended Forward* (*EF*) *implications*. These implications yield highly non-trivial relationships among signals in the circuit netlist, and when combined with the existing set of static logic implications consisting of direct, indirect and extended backward implications, results in a very powerful implication engine. Such an implication engine can be used for a variety of applications in Electronic Design Automation (EDA) domain such as Automatic Test Pattern Generation (ATPG) [Schulz 88, Larabee 92, Stephan 96], Logic Verification [Paul 00] , Logic Optimization [Ichihara 97, Kunz 97], Path Delay Testing [Heragu 97], Untestable Fault Identification [Zhao 97, Hsiao 00] etc. In this chapter we show the effectiveness of extended forward implications through additional untestable faults they help to identify.

5.1 Basic Idea

As explained in Chapter 2, extended backward implications are performed on *unjustified output specified gates* in the implication list of the *target gate*. These unjustified output specified gates are a subset of actual unjustified gates. The unjustified gates are the potential sites where learning can be performed to determine additional implications. In our approach we try to bridge the gap that was left by extended backward

implications. We perform learning on the unjustified gates that were not considered during computation of extended backward implications. Before proceeding further let us define a few terms pertaining to the potential sites for computing *extended forward implications.*

5.2 Definitions

- **EF learning sites:** The EF learning sites are the set of gates whose one or more inputs are specified, but these inputs/fanins do not determine the gate's output value.

  In other words, EF learning sites = [set of unjustified gates] – [set of unjustified output specified gates]

  Let us define the EF learning sites more precisely using mathematical notations.

  Consider a gate G with n inputs $l_1, l_2, ... l_n$

  Let $(l_i, w) \in impl[N, v]$ where $i=1,2,..k,$; $k<n$ and w is the non-controlling value of the gate G. Then, gate G is an *EF learning site* for the assignment (N, v).
- **EF learning set:** The set of gates G meeting the EF learning site criteria under (N, v) constitute the EF learning set for (N, v). This is represented as EF_learning_set $[N, v]$.

The main idea behind extended forward implications is to use the *EF learning set* for the assignment (N, v) to learn non-trivial logic implications.

5.3 Formulation of Extended Forward (EF) Implications

We define extended forward implications (EF) in the following way:

If $G \in$ EF_learning_set $[N, v]$, then two cases arise:

Case 1: If l_i is the only unspecified input of G then,

$EF = LogicSimulate\ (impl\ [N,\ v] \cup impl\ [l_i,\ 0]) \cap LogicSimulate\ (impl\ [N\ ,\ v] \cup impl\ [l_i,\ 1])$..... **(5.1)**

And, $impl\ [N,\ v] = impl\ [N,\ v] \cup EF$ ..... **(5.2)**

Case 2: If G has more than one unspecified input then,

$EF = LogicSimulate\ (impl\ [N,\ v] \cup impl\ [G,\ 0]) \cap LogicSimulate\ (impl\ [N\ ,\ v] \cup impl\ [G,\ 1])$ .....**(5.3)**

And, $impl\ [N,\ v] = impl\ [N,\ v] \cup EF$..... **(5.4)**

The motivation behind extended-forward implications is to push the envelope of implications of $(N,\ v)$ beyond the EF learning sites. For *Case 1,* this attempt to go beyond the EF-learning site is performed by trying both logic values for the unspecified input of G i.e. setting $l_i = 0$ and $l_i = 1$ and taking the intersection of the set of new logic implications learnt for each logic value. For *Case 2,* since more than one input of G are unspecified, it would be computationally expensive to try all possible value combinations for all unspecified inputs. So in this situation, we simulate both logic values for the gate output i.e. $G = 0$ and $G = 1$, to identify new implications. Since the underlying concept behind extended forward implication tries to *extend* the implications in the forward direction (bounded by the EF learning sites), hence the name.

In order to further understand the concept of extended-forward implications consider the example circuit shown in Figure 5.1:

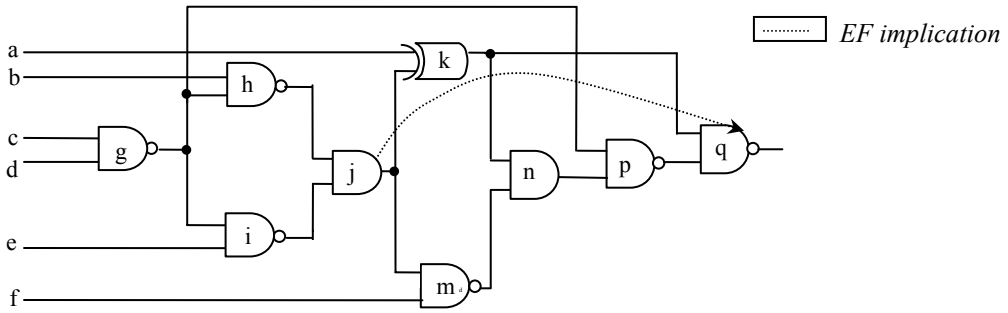


Figure 5.1 Example Circuit illustrating Extended Forward implications

Consider the implication set of (*g, 0*), given by *impl* [*g, 0*] = {(*g, 0*), (*h, 1*), (*i, 1*), (*j, 1*)}, where (*g, 0*) → (*j, 1*) is an indirect implication. By contrapositive law, (*g, 0*) → (*j, 1*) yields (*j, 0*) → (*g, 1*). Therefore, the implication set of (*j, 0*) is *impl* [*j, 0*]={(*j, 0*), (*m, 1*), (*g, 1*)}. From the implication (*j, 0*) we see that gate *k* is an EF learning site, similarly gates *n* and *h, i, p* are the EF learning sites due to implications (*m, 1*) and (*g, 1*), respectively. Thus, the EF_learning_set [*j, 0*] = {*k, n, h, i , p*}.

Consider one of the EF learning sites from the EF_learning_set[*j, 0*]. Let this be gate *p.* Now, gate *p* has only one unspecified input, namely gate *n*. This satisfies *Case 1* described above. Therefore using equation 5.1,

$EF = LogicSimulate\ (impl[j, 0] \cup impl[n, 0]) \cap LogicSimulate\ (impl[j, 0] \cup impl[n, 1])$

$\Rightarrow EF = \{(j, 0),\ (m, 1), (g, 1), (n, 0), (p, 1), (k, 0),$ **$(q, 1)$**$, (a, 0)\ \} \cap \{(j, 0),\ (m, 1), (g, 1),$

$(n, 1), (p, 0),$ **$(q, 1)$**$, (k, 1),\ (a, 1)\ \}$

$\Rightarrow EF = \{(j, 0),\ (m, 1), (g, 1), (q, 1)\}$

Hence, from equation 5.2,

$impl[j, 0] = impl[j, 0] \cup EF$

$\Rightarrow impl[j, 0] = \{(j, 0),\ (m, 1), (g, 1), (q, 1)\}$

Thus, (*j, 0*) → (*q, 1*) is a non-trivial *extended forward implication.*

The new class of implications which we term as *extended forward implications* strengthens the power of the existing implication engine consisting of direct, indirect and extended backward implications. The extended forward implications help us to deduce some highly non-trivial implication relationships among signals in the circuit netlist which can be helpful for variety of applications such as CEC, BMC, Logic Optimization, Untestable Fault Identification etc. In Section 5.4 we show the efficacy of this improved implication engine when it is applied to Untestable Fault Identification.

5.4 Experimental Results

We conducted all our experiments on a Pentium-4, 1.8GHz machine, with 512Mb of RAM and Linux as the operating system. ISCAS'85 and ITC'99 benchmarks were used for showing the effectiveness of extended forward implications for redundancy identification. In the discussion to follow, let the implication engine consisting of direct, indirect and extended forward implications be denoted by *ImpEng_ef,* the implication engine consisting of direct, indirect and extended backward implications be denoted by *ImpEng_eb* and the *new* enhanced implication engine consisting of all direct, indirect, extended backward and extended forward implications be denoted by *ImpEng_n*.

Table 5.1 shows the number of implications and number of constants obtained with *ImpEng_eb, ImpEng_ef* and *ImpEng_n.* Here, constants are the gate outputs in the circuit which are stuck at logic *0* or *1* permanently. We see form Table 5.1 that in almost all the cases (except *c1908* and *c3540*), the number of implications deduced by *ImpEng_ef* is more than *ImpEng_eb.* For circuit *c2670*, the highly non-trivial extended

Table 5.1 Number of Implications and Constants for *ImpEng_eb, ImpEng_ef* and *ImpEng_n*

Circuit	*ImpEng_eb*		*ImpEng_ef*		*ImpEng_n*	
	#Implications	#Constants	#Implications	#Constants	#Implications	#Constants
c1908	47440	0	47334	0	47484	0
c2670	62380	11	62940	15	63522	15
c3540	313123	1	311332	1	314495	1
c5315	107321	1	107868	1	108528	1
c6288	35100	17	51154	17	90051	17
c7552	306618	4	308706	4	313814	4
b01_c	956	0	964	0	964	0
b04_c	32570	6	32698	6	32794	6
b05_c	119844	57	130766	57	134956	57
b07_c	24394	0	25284	0	25322	0
b11_c	46108	7	50634	8	52265	8
b12_c	118108	0	119168	0	119628	0
b13_c	10095	3	10291	3	10361	3
b14_c	3002083	5	3002846	5	3062208	5

forward implications (*ImpEng_ef*) help to deduce a total of *15* constants compared to *11* reported by extended backward implications. Similarly, for circuit *b11_c ImpEng_ef* helps to deduce *8* constants as compared to *7* by *ImpEng_eb.* We see that difference in the number of implications between *ImpEng_ef* and *ImpEng_eb* is not much (except for *c6288*), implying that the two techniques have some amount of overlapping. The reason is the use of contrapositive law which helps to deduce some extended forward implications by *ImpEng_eb,* and some extended backward implications by *ImpEng_ef.* However, the additional non-trivial implications computed by extended forward implications can be very useful for redundancy identification as we will show in the next set of results. Also, we see from Table 5.1 that the total number of implications computed by *ImpEng_n* are always greater than those computed by *ImpEng_ef* or *ImpEng_eb* alone. This is quite obvious since the two techniques tend to complement each other to some extent. The maximum increase in number of implications was observed for *c6288*,

wherein the total number of implications more than doubled when extended forward implications were incorporated into the existing implication engine, *ImpEng_eb.*

In Table 5.2 we show the efficacy of our new improved implication engine when it is applied to Untestable Fault Identification. We implemented the Impossible Value Combination (IVC) algorithm proposed by Hsiao in [Hsiao 02] and described in Chapter 2 under Section 2.5.2. Table 5.2 gives the number of untestable faults when each of the implication engines *ImpEng_eb, ImpEng_ef* and *ImpEng_n* were used for redundancy identification using the IVC algorithm. Here the execution times reported are the time taken to build the implication graph only. The time taken to identify untestable faults is almost the same irrespective of the implication engine used and hence is not reported.

Table 5.2 Number of Untestable Faults and Execution Time with *ImpEng_eb, ImpEng_eb* and *ImpEng_eb*

Circuit	*ImpEng_eb*		*ImpEng_ef*		*ImpEng_n*	
	#Untestable faults	Time (secs.)	#Untestable faults	Time (secs.)	#Untestable faults	Time (secs.)
c1908	9	0.47	9	0.60	9	1.01
c2670	75	0.78	83	1.02	83	1.68
c3540	131	3.37	137	5.03	137	7.60
c5315	58	2.19	59	2.80	59	4.46
c6288*	34	1.00	34	1.57	34	2.02
c7552	64	9.69	67	12.42	67	20.34
b04_c	8	0.25	8	0.37	8	0.61
b05_c	475	2.17	492	3.02	492	4.89
b07_c	0	0.13	0	0.24	0	0.31
b11_c	61	0.31	65	0.48	65	0.72
b13_c	26	0.05	26	0.10	26	0.12

* *the circuit has a total of 34 redundant faults*

We see from Table 5.2 that the time taken by *ImpEng_ef* is somewhat greater than the time taken by *ImpEng_eb*. Also, the time taken by *ImpEng_n* is approximately the sum of the time taken by *ImpEng_eb* + *ImpEng_ef* together. Comparing the number of untestable faults in column 2 and column 4, we see that for a good number of circuits we

get an increase in number of untestable faults. For example, for the circuits *c2670* and *c7552*, the number of untestable faults increase by *8* and *3,* respectively. Similarly, for *b11_c* the number of untestable faults increases from *61* to *65*. For the circuit *c5315*, all possible redundant faults (*59*) were identified with *ImpEng_ef* alone. Although, for the reported circuits we observed that the number of untestable faults obtained with *ImpEng_ef* and the new improved implication engine *ImpEng_n* (with both extended backward and extended forward implications) are the same, it does not mean that extended backward implications are not useful. We believe that the new strengthened implication engine *ImpEng_n* might be beneficial for other EDA applications such as path delay testing [Heragu 97], logic optimization [Ichihara 97] etc., which have not been explored here.

CHAPTER 6

FORMALIZING GLOBAL LEARNING FOR SIMPLIFICATION OF A GENERIC CNF FORMULA

A large variety of problems in EDA domain such as logic synthesis, equivalence checking, bounded model checking, ATPG etc. reduce to the satisfiability (SAT) problem. Also, many problems in Artificial Intelligence (AI) can be framed as SAT instances. The SAT problem is formulated in Conjunctive Normal Form (CNF) commonly known as the Product of Sum (POS) form, and the resulting CNF instance is then given to the SAT solver for processing. There are different classes of SAT solvers known, each suited for some specific application. While some of the CNF instances might be extremely hard for one class of SAT-solvers, the same set of CNF instances might be pretty easy for another class of SAT-solvers.

In this chapter, we propose a preprocessing technique that tries to simplify a generic CNF instance, such that the resulting formula is easier for any SAT-solver to solve. The preprocessing technique is independent of the class of the SAT-solver used. Also, if the original CNF formula is a representation of two-level logic circuit, this simplification will yield a minimized two-level logic circuit. The basis of this simplification is the suite of Lemmas and Theorems we proposed in Chapter 3 which were based on *implication reasoning*. In Chapter 3, we worked on the circuit netlist (built from Boolean gates) to determine static logic implications consisting of direct, indirect

and extended backward implications. The Lemmas and Theorems mapped these implications on to the CNF formula, and showed that the clauses added through static logic implications will preserve the accuracy of the CNF formula.

Since the Theorems presented in Chapter 3 can deduce non-trivial clauses, which are much more powerful than what static logic implications consisting of direct, indirect and extended backward implications can deduce, we carried out a full implementation of these Theorems to show their effectiveness. Also, the Theorems work on a generic CNF formula and unlike the static logic implications do not require any circuit structure. Thus, these theorems based on *implication reasoning* can even be applied to problems which do not have a circuit representation, but can be formulated as a SAT instance represented by a CNF formula (example AI planning problems). We also present some additional theorems in section 6.2 that were not discussed in Chapter 3, and which further help to simplify the CNF formula.

In the Experimental results, we compare our preprocessing tool with Hypre [Bacchus 03] and show that for a large number of instances our approach outperforms Hypre and leads to a greater simplification of the CNF formula.

6.1 Review of Lemmas and Theorems on Implication Reasoning

In this section we review the set of Lemmas and Theorems proposed in Chapter 3 which formalize global learning. These Lemmas and Theorems based on implication reasoning form the basis of our preprocessing, and help to deduce additional clauses (unit clauses and two-literal clauses) which can simplify the CNF formula. Pertaining to each

of the Lemmas and Theorems, we also give examples such that the readers can comprehend the concepts quite easily.

Lemma 1. Given a CNF formula Φ, if $(y, 1) \in$ BCP $(x, 1)$, then the clause $(\neg x \vee y)$ is an implicate of Φ.

Proof: See Chapter 3 (Section 3.3).

Theorem 1. Given a CNF formula Φ, if $(y_i, 1) \in$ BCP $(x, 1)$, $i = 1, 2, \ldots n$, then the clauses $(\neg x \vee y_i)$ are implicates of Φ.

Proof: See Chapter 3 (Section 3.3).

Example illustrating Theorem 1

Consider the CNF formula Φ shown below:

$\omega 1 = (\neg f \vee \neg e \vee \neg c)$, $\omega 2 = (f \vee g)$,

$\omega 3 = (\neg f \vee \neg d \vee \neg h)$, $\omega 4 = (f \vee h)$,

$\omega 5 = (\neg g \vee \neg h \vee i)$

Now, let us do Boolean Constraint Propagation with f set to value 0. When $f = 0$, applying unit clause rule on ω2 and ω4, yields $g = 0$ and $h = 0$, respectively. Now, applying unit clause rule on ω5 yields $i = 0$. Therefore, BCP $(f, 0) = \{(f, 0), (g, 1), (h, 1), (i, 1)$

Since the clauses $(f \vee g)$ and $(f \vee h)$ are already present, we can add up the clause **$(f \vee i)$.**

Lemma 2: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if $(y, 1) \in [\cap^{n}_{i=1} \text{BCP}(l_k, 1)]$, then $(y, 1)$ will be a necessary assignment of Φ.

For the original CNF formula Φ to be satisfied, every clause $\omega \in \Phi$ needs to be satisfied. Clause ω can be satisfied by setting any of its literals to logic 1. Therefore, any common assignment obtained by setting each of the literals in ω to logic 1, will be a necessary assignment.

Proof: See Chapter 3 (Section 3.3).

Example illustrating Lemma 2

Consider the CNF formula Φ shown below:

$\omega 1 = (a \vee \neg c)$, $\omega 2 = (b \vee \neg c)$,

$\omega 3 = (a \vee b)$, $\omega 4 = (\neg a \vee \neg b)$

Now, let us consider the satisfiability of clause ω4. ω4 can be satisfied either by setting $a=0$ or by setting $b=0$.

Setting $a=0$, we get BCP $(a, 0) = \{(a, 0), (b, 1), (c, 0)\}$

Setting $b=0$, we get BCP $(b, 0) = \{(b, 0), (a, 1), (c, 0)\}$

Therefore, BCP $(a, 0) \cap$ BCP $(b, 0) = \{(c, 0)\}$

Hence, using Lemma 2 ***(c, 0)*** is the necessary assignment for the CNF formula Φ to be satisfied.

Lemma 3: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if under the assignment $(x, 0)$, the literals $l_1, l_2 \ldots. l_j$ $(j < n)$ are implied to 0, and if $(y, 1) \in [\cap^{n}_{k=j+1} \text{BCP}(l_k = 1 \text{ and } x = 0)]$, then $(x \vee y)$ will be an implicate of Φ.

This Lemma is an extension of Lemma 2 and states that if the current assignment ($x, 0$) implies the literals $l_1, l_2 \ldots. l_j$ ($j < n$) of ω to logic 0, then the common assignment ($y, 1$) obtained by setting each of the remaining literals of ω to 1, together with the current assignment ($x, 0$) will result in an implicate ($x \vee y$) of Φ.

Proof: See Chapter 3 (Section 3.3).

Theorem 2: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if under the assignment ($x, 0$), the literals $l_1, l_2 \ldots. l_j$ ($j < n$) are implied to 0, then for every $(y_i, 1) \in [\cap^{n}_{k=j+1}(\text{BCP } (l_k = 1 \text{ and } x = 0)]$, $i = 1, 2, ...m$, $(x \vee y_i)$ is an implicate of Φ.

Proof: See Chapter 3 (Section 3.3).

Example illustrating Theorem 2

Consider the CNF formula Φ shown below:

$\omega 1 = (f \vee e)$, $\omega 2 = (f \vee h)$, $\omega 3 = (f \vee g)$,

$\omega 4 = (\neg g \vee \neg h \vee i)$, $\omega 5 = (\neg e \vee a \vee b)$,

$\omega 6 = (\neg j \vee \neg a \vee \neg i)$, $\omega 7 = (j \vee \neg m)$,

$\omega 8 = (\neg h \vee \neg b \vee \neg k)$, $\omega 9 = (k \vee \neg m)$

Let us make the assignment $f=0$. Therefore, BCP ($f, 0$) = {($f, 0$), (e, 1), ($h, 1$), ($g, 1$),($i, 1$)}

Now, consider the satisfiability of clause $\omega 5 = (\neg e \vee a \vee b)$. We see that under the assignment $f=0$, the literal $\neg e$ is implied to 0. Therefore, for the clause ω5 to be satisfied either a or b needs to be equal to 1.

Setting $a=1$, under the assignment $f=0$, we get BCP ($a=1$ and $f=0$) = {($f, 0$), (e, 1), ($h, 1$), ($g, 1$), ($i, 1$), ($a, 1$), ($j, 0$), ($m, 0$)}

Setting $b=1$, under the assignment $f=0$, we get BCP ($b=1$ and $f=0$) = {($f, 0$), (e, 1), ($h, 1$), ($g, 1$), ($i, 1$), ($b, 1$), ($k, 0$), ($m, 0$)}

Thus, { BCP ($a=1$ and $f=0$) } $\cap$ { BCP ($b=1$ and $f=0$) } = {($f, 0$), (e, 1), ($h, 1$), ($g, 1$), ($i, 1$), ($m, 0$)}

Therefore, using Theorem 2 the clauses $(f \vee i)$ and $(f \vee \neg m)$ can be added.

Note that clauses ($f \vee e$), ($f \vee h$) and ($f \vee g$) are already present as ω1, ω2 and ω3 and hence not added.

6.2 New Theorems on Implication Reasoning

In this section, we introduce a few more theorems based on the analysis of CNF formula using *implication reasoning*. We continue to *number* the theorems from where we left in section 6.1.

Theorem 3: Given a CNF formula Φ, for any clause $\omega = (l_1 \vee l_2 \vee ..... l_n) \in \Phi$, if under the assignment ($x, 0$), the literals $l_1, l_2 \ldots. l_j$ ($j < n$) are implied to 0, and if BCP (l_m =1 and $x=0$) yields a conflict, such that $m \in j+1, j+2, \ldots.. n,$ then $(x \vee \neg l_m)$ is an implicate of Φ.

Proof: We are given that the assignment ($x, 0$) causes the following:

- $l_1, l_2 \ldots\ldots, l_j$ are implied to 0, and …….. (*1*)
- BCP ($x=0$ and $l_m=1$) results in at least one of the clauses of Φ evaluating to *0* …….(*2*)

We continue the proof by contradiction:

Assume that $(x \vee \neg l_m)$ is not an implicate of Φ. Then, two cases can arise:

1. (l_m, *1*) ∈ BCP (*x, 0*). This according to the given condition (*2*) will cause one of the clauses of Φ to evaluate to *0*. Hence, (l_m, *1*) ∈ BCP (*x, 0*) is not possible.
2. (*x, 0*) ∈ BCP (l_m, *1*). This according to given condition (*2*) will cause one of the clauses of Φ to evaluate to *0*. Hence, (*x, 0*) ∈ BCP (l_m, *1*) is not possible.

Since the above two cases are not possible, ($x \vee \neg l_m$) is an implicate of Φ.

Theorem 4: Given a CNF formula Φ, for any clause ω = ($l_1 \vee l_2 \vee$ ..... l_n) ∈ Φ, if under the assignment (*x, 0*), the literals l_1, l_2 …. l_j ($j < n$) are implied to 0, and if BCP (l_k =*1* and *x* =*0*) yields a conflict ∀ *k* = *j+1, j+2, …… n,* then (*x, 1*) is a necessary assignment of Φ.

Proof: We are given that the assignment (*x, 0*) causes the following:

- $l_1, l_2 \ldots\ldots, l_j$ are implied to 0, and …….. (*i*)
- BCP(*x*=*0* and l_k=*1*) ∀ *k* = *j+1, j+2, …… n,* results in at least one of the clauses of Φ evaluating to *0,* ……..….(*ii*)

We continue the proof by contradiction. Suppose that (*x, 1*) is not a necessary assignment. In other words, there exists a satisfying assignment to the CNF formula with (*x, 0*).

Under the assignment (*x, 0*), the following are true:

- BCP(*x*=*0* and l_{j+1}=*1*) yields a conflict, therefore the implicate ($x \vee \neg l_{j+1}$) will follow from Theorem 3……...……(*1*)
- BCP(*x*=*0* and l_{j+2}=*1*) yields a conflict, therefore the implicate ($x \vee \neg l_{j+2}$) will follow from Theorem 3 ……...……..(*2*)
- ……

- BCP ($x=0$ and $l_n=1$) yields a conflict, therefore the implicate ($x \vee \neg l_n$) will follow from Theorem 3……(*n-j*)

Using (*1*) to (*n-j*), we see that under (*x, 0*) , the literals l_{j+1}, l_{j+2}, ... l_n will be implied to *0*. Also, from (*i*) we already know that (*x, 0*) causes $l_1, l_{2\ldots\ldots}, l_j$ to be implied to *0*. Hence, the clause ω will evaluate to *0,* causing the CNF formula Φ to become unsatisfiable. Thus, our assumption is false and the assignment (*x, 0*) is not possible. Therefore, (*x, 1*) is a necessary assignment.

6.3 Efficacy of the Theorems

The Theorems and Lemmas presented in Section 6.1 and Section 6.2 are highly effective and help to deduce clauses yielding non-trivial relationships among the variables in the CNF formula. These relationships in turn help to simplify the CNF formula by reducing the number of variables and/or reducing the number of clauses. The following are some of the applications of the presented Theorems:

- They help to deduce the necessary assignments or *unit literals*.
- They help to identify *equivalent literals.* If the CNF formula after augmentation has two clauses of the form ($x \vee \neg y$) and ($\neg x \vee y$), it implies $x \equiv y$ and we can perform equality reduction, similar to the one discussed in Chapter 1, Section 1.1.
- They help to identify *complement literals.* If the CNF formula after augmentation has two clauses of the form ($\neg x \vee \neg y$) and ($x \vee y$), it implies $x \equiv \neg y$ and we can further simplify the CNF formula.

- The help to deduce other two-variable implication relationships which are not as strong as unit, equivalent or complement literals. For example, if we deduce a clause (x $\vee \neg$y), then the decision $x = 0 \rightarrow y = 0$ and the decision $y = 1 \rightarrow x = 1$. This is less stronger than *x* and *y* being equivalent.

6.4 Implementation Issues

The number of clauses that can be deduced using the above theorems can be prohibitively large and hence time consuming. Also, some clauses are not as useful as others in terms of their deduction power. As a result, these additional *not so useful clauses* can become an overhead for the preprocessor and the SAT-solver. The problem of finding which clauses are useful and should be added to the existing CNF database has been an area of research for a lot of preprocessing tools [Lynce 03, Bacchus 03].

We follow an approach of selectively adding the clauses based on *binary resolution*. Let us try to explain our approach by considering the CNF formula Φ given below:

$\omega 1 = (f \vee h)$, $\omega 2 = (f \vee g)$,

$\omega 3 = (\neg g \vee \neg h \vee i)$, $\omega 4 = (\neg i \vee j)$

Now, BCP $(f, 0) = \{(f, 0), (h, 1), (g, 1),$ **$(i, 1), (j, 1)$** $\}$

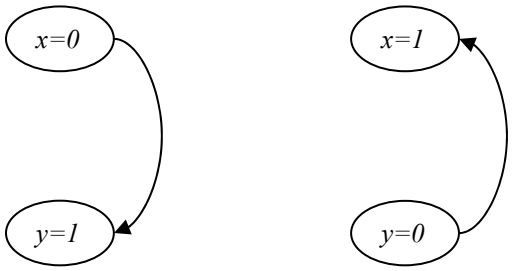
So, from Theorem 1, we can add up the clauses $(f \vee i)$ and $(f \vee j)$. However, in our implementation we only add up the clause $(f \vee i)$, and do not add the clause $(f \vee j)$. The reason is that once we add up the clause $(f \vee i)$, the clause $(f \vee j)$ can be easily obtained

through one-step binary resolution on $(f \vee i)$ and $(\neg i \vee j)$. Hence, adding the clause $(f \vee i)$ reduces the effectiveness of the clause $(f \vee j)$. We call this approach as *selective learning* and used it during the implementation of other Theorems as well. It was observed in our experiments that this technique helped to reduce the number of clauses by almost *50%* without much sacrifice in the deduction power of the clauses.

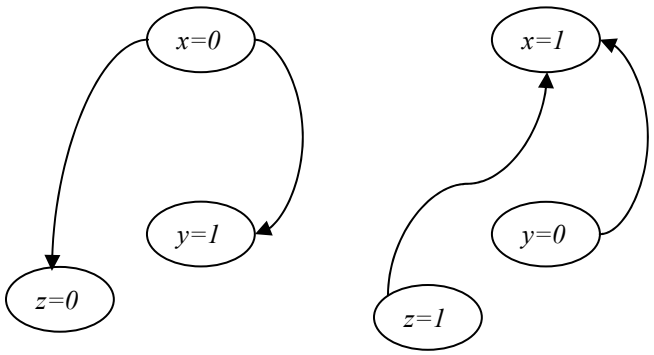
The addition of the clauses using *selective learning* is easier said than done. This was one of the difficult problems which we tackled by building an *implication graph* on the fly. This implication graph is similar to the one described in Chapter 2, under Section 2.1.4. Below, we illustrate how the implication graph is built.

- Associate a node with every literal in the CNF formula.
- For every 2-literal clause added, modify the implication graph on the fly.

  For example, if a clause $\omega = (x \vee y)$ is deduced, add the edges $x=0 \rightarrow y=1$ and $y=0 \rightarrow x=1$ as shown below.



- Similarly, if a clause $\omega = (x \vee \neg z)$ is deduced, add up the edges $x=0 \rightarrow z=0$ and $z=1 \rightarrow x=1$, as shown below



Now, whenever a literal (*parent node*) is chosen for implication reasoning on clauses, find the nodes in the transitive closure of this parent node from the implication graph and mark them. Then the parent node (literal) along with each of the marked nodes are already in CNF database as two-literal clauses and need not be added. This approach helped us to significantly improve the performance of our preprocessor.

6.5 Experimental Results

We conducted all our experiments on a Pentium-4, 1.8GHz machine, with 512Mb of RAM and Linux as the operating system. The presented Lemmas and Theorems were implemented in C++ in a preprocessing tool called CAIR (CNF Analysis using Implication Reasoning). We used the CNF instances from ISCAS'85 Benchmarks, FVP UNSAT 1.0 [Velev], and miters from [Silva] to show the effectiveness of our tool. The experimental results are given in Table 6.1.

Table 6.1 gives the name of the CNF instance and the corresponding simplification obtained by using the tools Hypre [Bacchus 03] and CAIR + Hyper. The simplification is in terms of the number of variables and clauses obtained after applying Hypre and CAIR + Hyper, respectively, to the original CNF instance. The preprocessor Hypre can either prove the CNF instance to be unsatisfiable or generate a simplified CNF formula. The term UNSAT in Table 6.1 indicates that the CNF instance was proven

Table 6.1 CNF formula simplification with Hypre and CAIR + Hypre

CNF instance	Hyper		CAIR +Hyper	
	#Variables	#Clauses	#Variables	#Clauses
c2670	686	1922	677	1854
c3540	820	3055	818	3062
c5315	1397	4608	1395	4610
c7552	1936	6386	1931	6473
1dlx_c_mc_ ex_bp_f	707	3575	663	3332
2dlx_ca_mc _ex_bp_f	2809	24652	UNSAT	
2dlx_cc_mc_ex_bp_f	3727	35920	3537	39153
c2670_miter	778	2727	UNSAT	
c3540_miter	1165	4973	UNSAT	
c5315_miter	1606	6701	UNSAT	
c7552_miter	2174	9203	2048	8904
c2670_bug	805	2708	753	2574
c3540_bug	1162	4965	907	4283
c5315_bug	1796	7698	1671	7258
c7552_bug	2277	9692	2177	9565

unsatisfiable without generating the simplified formula. Analyzing Table 6.1, we see that for all the cases CAIR + Hyper leads to a greater reduction in the number of variables than Hypre alone. Also, for most of the cases the number of clauses obtained with CAIR + Hypre are much less than those obtained with Hypre alone. For example, in case of *c3540_bug* the number of variables and clauses obtained after applying Hyper alone were *1162* and *4965*, respectively. On the other hand, after applying CAIR + Hypre to the original CNF instance the number of variables and clauses reduced to *907* and *4283*, respectively. Also, we see that for 4 of the instances namely *1dlx_c_mc_ ex_bp_f, c2670_miter , c3540_miter* and *c5315_miter,* Hypre alone could not prove that the CNF formulae were unsatisfiable. However, using CAIR as a preprocessor to Hypre helped to deduce some highly non-trivial relationships among the CNF variables, which in turn assisted Hypre to prove the unsatisfiability immediately.

It should be mentioned that our preprocessing tool CAIR is two to three times slower than Hypre, because the worst complexity of the Theorems 2-4 is O(*mn*); *n* is the number of variables and *m* is the number of clauses. However, we believe that for the hard instances when such a simplified CNF formula would be given to the SAT-solver for processing, the preprocessor time would not be much of an overhead, and the performance improvement would be significant.

In order to build an efficient and robust preprocessing tool we plan to reduce the computation complexity of CAIR as a part of our future work.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

We presented a novel method of augmenting the original CNF formula with static logic implications. Two-literal clauses resulting from indirect and extended backward implications were quickly computed and added to the existing CNF database. For sequential circuits, these clauses spanned multiple time-frames and were quickly computed without unrolling the circuit, making our method very cost effective. These added clauses served as constraints and helped to induce global structural information throughout the CNF formula of the circuit-under-verification. This in turn aided the SAT-solver in the search process. Experimental results for combinational equivalence checking (CEC) showed that irrespective of the state-of-the-art SAT-solver used, we achieved more than one order of magnitude speedup for most of the instances, with the actual speedup ranging from 1.06X to 827.35X. Comparison with other preprocessing techniques like Hypre [Bacchus 03] corroborated the fact that the clauses obtained using our method are much more powerful and reduce the SAT instance complexity considerably.

Although, efforts have been made in the past to improve SAT-based Bounded Model Checking (BMC) using local structural relationships, we believe that ours is the first approach of its kind to use global structural relationships spanning time-frame

boundaries. We showed that using our novel technique we achieved speedups of up to 148.98X over the conventional SAT-based Bounded Model Checking approach which is quite noteworthy.

We also introduced a non-trivial global learning technique resulting in a new class of implications termed as *extended forward* (*EF*) *implications.* These implications when combined with the existing set of static logic implications consisting of direct, indirect and extended backward implications, resulted in a very powerful implication engine. Such an implication engine can be used for a large variety of applications in Electronic Design Automation (EDA) domain such as Automatic Test Pattern Generation (ATPG), Logic Verification, Logic Optimization, Path Delay Testing and Untestable Fault Identification.

Lastly, we presented and implemented a suite of lemmas and theorems that formalized global learning, and helped to simplify a generic CNF formula. Comparison of our preprocessing tool CAIR with the publicly available preprocessor Hypre [Bacchus 03] showed that the clauses deduced through our *implication reasoning* technique are highly non-trivial and powerful, and that they allow greater simplification of the CNF formula. The simplified CNF formula when given to the SAT-solver for processing is bound to improve its performance.

7.2 Future Work

In this Thesis we concentrated on learning global relationships involving at most two signals or variables. However, we believe that there is a lot of potential, if we can extract non-trivial relationships among a group of signals in the circuit netlist, or among a

group of variables in the CNF formula. Such global multi-signal relationships can in turn be used for various applications that have/have not been explored in this thesis. The obvious bottleneck in computing multi-signal relationships is the time complexity and what relationships to look for. We plan to work on this as a part of our future work.

The implementation of the Theorems presented in Chapter 6 is computationally expensive for *easier to solve* instances. To counter this, we plan to use some kind of heuristics or probability based measures for choosing the variables/clauses during implication reasoning. We also plan to overcome this drawback by indulging in some kind of variable partitioning so that not all the clauses and variables are considered during the implication reasoning process. All this will help in building a highly efficient and robust preprocessor.

References

[Agarwal 95] V. D. Agarwal and S. T. Chakradhar, "Combinational ATPG Theorems for Identifying Untestable Faults in Sequential Circuits," *IEEE Trans. Computer-Aided Design,* vol. 14, no. 9, Sept. 1995, pp. 1155-1160.

[Bacchus 02] F. Bacchus, "Enhancing Davis Putnam with Extended Binary Clause Recording", In *Proceedings of National Conference on Artificial intelligence* (*AAAI-2002*), August 2002, pp. 613-619.

[Bacchus 03] F. Bacchus and J. Winter, "Effective Preprocessing with Hyper-Resolution and Equality Reduction", In *Lectures notes in Computer Science, Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003,* Volume 2919 / 2004, pp. 341-355.

[Biere 99] A. Biere, A. Cimatti, E. Clarke and Y. Zhu, "Symbolic Model Checking Without BDDs", In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*) *Conference*, March 1999, pp. 193-207.

[Boppana 99] V. Boppana, S. P. Rajan, K. Takayama, M. Fujita, "Model Checking Based On Sequential ATPG", In *Proceedings of Computer Aided Verification* (*CAV*)*,* Lecture Notes in Computer Science, 1633 Springer 1999, pp. 418-430 .

[Brglez 85] F. Brglez and H. Fujiwara, "A Neural Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," In *Proceedings of International Symposium on Circuits and Systems (ISCAS) Conference*, June 1985, pp. 663-698.

[Brglez 89] F. Brglez, D. Bryan and K. Kozminski, "Combinational Problems of Sequential Benchmark Circuits," In *Proceedings of International Symposium on Circuits and Systems* (*ISCAS*) *Conference*, June 1989, pp. 1929-1934.

[Burch 90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} State and Beyond.", In *Proc. Logic Computer Science (LICS),* June 1990, pp. 428-439.

[Cabodi 02] G. Cabodi, P. Camurati and S. Quer, "Can BDDs compete with SAT solvers on Bounded Model Checking?", In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*)*,* June 2002, pp. 117-122.

[Cabodi 03] G. Cabodi, S. Nocco and S. Quer, " Improving SAT-based Bounded Model Checking by Means of BDD-based Approximated Traversals", In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*)*,* 2003, pp. 898-903.

[Chakradhar 93] S.T. Chakradhar and V. D. Agarwal, "A Transitive Closure Algorithm for Test Generation", *IEEE Transactions on Computer Aided Design*, 1993, pp. 1015- 1028.

[Clarke 86] E. Clarke, E. A. Emerson and A. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications", In *ACM Trans. Programming Languages and Systems*, Vol. 1, no. 2, 1986, pp. 244–263.

[Clarke 02] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[Corno 00] F. Corno, M. Sonza Reorda and G. Squillero "RT-Level ITC 99 Benchmarks and First ATPG Results", In *IEEE Design and Test of Computers,* July-August 2000, pp. 44-53.

[Davis 62] M. Davis, G. Longemann and D. Loveland "Machine Program for Theorem Proving", *Communications of the ACM,* Vol. 5, 1962, pp. 394-397.

[Gelder 93] A. Van Gelder and Y.K. Tsuji, "Satisfiability Testing with More Reasoning and Less Guessing", In *Second DIMACS Implementation Challenge,* American Mathematical Society, editors D.S. Johnson and M. A. Trick, 1993.

[Goldberg 02a] E. Goldberg and Y. Novikov, "Berkmin: A Fast and Robust SAT Solver", In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*), March 2002, pp.142-149.

[Goldberg 02b] E. Goldberg and Y. Novikov, BerkMin561, http://eigold.tripod.com/BerkMin

[Gupta 03] A. Gupta, M. Ganai, C. W. Yang and P. Ashar, "Learning From BDDs in SAT-based Bounded Model Checking", In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2003, pp. 824-829.

[Heragu 97] K. Heragu, J.H. Patel, V.D. Agarwal, "Fast Identification of Untestable Delay Faults using Implications", In *Proceedings of International Conference on Computer-Aided Design* (*ICCAD*), Nov. 1997, pp. 642 – 647.

[Hsiao 02] M. S. Hsiao, "Maximizing Impossibilities for Untestable Fault Identification," In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*), March 2002, pp. 949-953.

[Ichihara 97] H. Ichihara, K. Kinoshita, "On acceleration of Logic Circuits Optimization using Implication Relations", In *Proceedings of Asian Test Symposium*, Nov. 1997, pp. 222 - 227

[Iyer 96a] M. A. Iyer and M. Abramovici, "FIRE: a Fault Independent Combinational Redundancy Algorithm," In *IEEE Transactions of Very Large Scale Integration* (*VLSI*) *Systems,* Volume 4, Issue 2, June 1996, pp. 295-301.

[Iyer 96b] M.A. Iyer, D.E. Long and M. Abramovici, "Identifying Sequential Redundancies Without Search," In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 1996, pp. 457-462.

[Kuehlmann 01] A. Kuehlmann, M.K. Ganai and V. Paruthi, "Circuit-Based Boolean Reasoning", In *Proceedings of IEEE/ACM Design Automation Conference* (*DAC*), June 2001, pp. 232-237.

[Kunz 92] W. Kunz and D.K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for the Test Generation in Digital Circuits", In *Proceedings of International Test Conference* (*ITC*), September 1992, pp. 816-825.

[Kunz 93] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", In *Proceedings of IEEE/ACM International Conference on Computer Aided Design* (*ICCAD*), November 1993, pp. 538-543.

[Kunz 97] W. Kunz, D. Stoffel, and P. R. Menon, "Logic Optimization and Equivalence Checking by Implication Analysis", In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Volume: 16 , Issue: 3, March 1997, pp.266 – 281.

[Larabee 92] T. Larabee, "Test Pattern Generation using Boolean Satisfiability", In *IEEE Transactions on Computer Aided Design,* Vol. 11, January 1992, pp. 4-15

[Li 00] C. Min Li, "Integrating Equivalency Reasoning into Davis-Putnam Procedure", In *Proceedings of National Conference of Artificial Intelligence* (*AAAI-2000*), July 2000, pp. 291-296.

[Lu 03a] F. Lu, Li-C. Wang, K-T. Cheng and R. C-Y Huang, "A Circuit SAT Solver with Signal Correlation Guided Learning", In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*), March 2003, pp. 892-897.

[Lu 03b] F. Lu, Li-C. Wang, K.- T. Cheng, J. Moondanos and Z. Hanna, " A Signal Correlation Guided ATPG Solver and its Applications for Solving Difficult Industrial Cases", In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2003, pp. 436-441.

[Lynce 03] I. Lynce and J.P. Marques-Silva, "Probing-Based Preprocessing Techniques for Propositional Satisfiability", In *15th IEEE International Conference on Tools with Artificial Intelligence,* November 2003, pp. 105-110.

[McMillan 93] K. L. McMillan *Symbolic Model Checking: An Approach to State Explosion Problem*. Kluwer Academic publishers, 1993.

[Moskewicz 01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver", In *Proceedings of ACM/IEEE Design Automation Conference* (*DAC*), June 2001, pp. 530-535.

[Novikov 03] Y. Novikov, "Local Search for Boolean Relations on the Basis of Unit Propagation", In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*), March 2003, pp. 810 -815.

[Paul 00] D. Paul, M. Chatterjee and D. K. Pradhan, "VERILAT: Verification Using Logic Augmentation and Transformations", In *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 19, no. 9, Sept. 2000.

[Peng 00] Q. Peng, M. Abramovici and J. Savir, "MUST: Multiple-Stem Analysis for Identifying Sequentially Untestable Faults," In *Proceedings of International Test Conference* (*ITC*) , 2000. pp. 839-846.

[Rajski 90] J. Rajski and H. Kox, "A Method to Calculate Necessary Assignments in ATPG," In *Proceedings of International Test Conference* (*ITC*),1990, pp. 25-34

[Reddy 99] S. M. Reddy, Irith. Pomeranz, X. Lim and Nadir Z. Basturkmen, "New Procedures for Identifying Undetectable and Redundant Faults in Synchronous Sequential Circuits," In *Proceeding of VLSI Test Symposium*, April 1999, pp. 275-281.

[Ryan 03] L. Ryan, Siege v4, 2003 http://www.cs.sfu.ca/~loryan/personal

[Saab 03] D. G. Saab, J. A. Abraham and V. M. Vedula, "Formal Verification Using Bounded Model Checking: SAT versus Sequential ATPG Engines", In *Proceedings of VLSI Design Conference,* 2003, pp. 243-248.

[Schulz 88] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", In *IEEE Transactions on Computer Aided Design,* Vol. 7, January 1988, pp. 126-137.

[Schulz 89]M. H. Schulz and E. Auth, "Improved deterministic test pattern generation with applications to redundancy identification", In *IEEE Transactions on Computer-Aided Design.*, vol. 8, July 1989, pp. 811–816.

[Silva 99a] J. P. Marques-Silva and K. A. Sakallah, " GRASP: A Search Algorithm for Propositional Satisfiability", In *IEEE Transaction on Computers,* Vol. 48, May 1999, pp. 506-521.

[Silva 99b] J. P. Marques Silva and L. Guerra E Silva, “Solving Satisfiability in Combinational Circuits using Backtrack Search and Recursive Learning”, In *Proceedings of XII Symposium on Integrated Circuits and System Design,* October 1999, pp. 192-195.

[Silva 99c] J. P. Marques-Silva and T. Glass,“ Combinational Equivalence Checking using Satisfiability and Recursive Learning”, In *Proceedings of Design, Automation and Test in Europe Conference* (*DATE*), March 1999, pp. 145-149.

[Silva] J. P. Marques-Silva , ftp://algos.inesc.pt/pub/benchmarks/cnf/equiv-checking /MITERS.

[Stephan 96] P. Stephan, R.K. Brayton and A. L. Sangiovanni Vincentelli, “Combinational Test Generation using Satisfiability”, In *IEEE Transactions on Computer Aided Design,* Vol. 15, September 1996, pp. 1167-1176.

[Velev] M.N. Velev, FVP-UNSAT.1.0. Available from: http://www.ece.cmu.edu/~mvelev.

[ZhangH 97] H. Zhang, “SATO: An Efficient Propositional Prover” In *Proceedings of International Conference on Automated Deduction*, vol. 1249, *LNAI*, July 1997, pp. 272-275.

[ZhangL 01] L. Zhang, C. Madigan, M. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", Proceedings of International Conference on Computer Aided Design (*ICCAD*), November 2001, pp. 279-285.

[Zhao 97] J. Zhao, M. Rudnick and J. Patel, “Static Logic Implication with Application to Fast Redundancy Identification”, In *Proceedings of VLSI Test Symposium* (*VTS*), April 1997, pp. 288-293.

[Zhao 01] J. Zhao, J. A. Newquist and J. Patel, “A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-cycle Redundancy Identification”, In *Proceedings of VLSI Design Conference,* January 2001, pp. 163-169.

VITA

Rajat Arora was born in Srinagar, the capital city of Jammu and Kashmir, India. He did his schooling partly from Srinagar and partly from Roorkee, India. He joined Punjab Engineering College, Chandigarh, India in 1998, to obtain technical education in the area of Electrical Engineering. After graduating with a Bachelor's Degree in May 2002, he joined Virginia Polytechnic Institute and State University in Fall 2002 to pursue a Masters degree in the Bradley Department of Electrical and Computer Engineering. He joined Dr. Michael Hsiao and his research group in January 2003 and has since then been involved in research related to Design Verification and VLSI Testing. He recently got a job in Cadence Design Systems, as a Member of Technical Staff (R & D Department) in Formal Verification in San Jose, CA. Rajat's hobbies include watching cricket, movies, swimming and listening to music.