A New Data Structure for Complete Implication Graph with Application for Static Learning

Emil Gizdarski[1]
Department of Computer Systems
University of Rousse
7017 Rousse, Bulgaria
egizdarski@ieee.org

Hideo Fujiwara
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
fujiwara@is.aist-nara.ac.jp

Abstract. In this paper we analyze learning techniques based on the Boolean satisfiability method and find that static indirect $\wedge$-implications and the super gate extraction are useful for increasing the precision of low complexity learning procedures. We propose a new data structure for the complete implication graph that allows efficient processing of the static indirect $\wedge$-implications. We show that by deriving and performing the static indirect $\wedge$-implications, some hard-to-detect static indirect implications can be easily found during static learning. In addition, the static indirect $\wedge$-implications can be used to perform (without spare operations) some dynamic indirect implications during branch and bound search and dynamic learning. In this way, the new data structure of the complete implication graph increases efficiency and precision of both static and dynamic learning as well as branch and bound search. We implement this data structure in two implicit static learning procedures. Experimental results for the ISCAS'85 and ISCAS'89 benchmark circuits demonstrate the efficiency and precision of the implemented static learning procedures as well as the efficiency of the new data structure of the complete implication graph. We expect that the contribution of the new data structure will be more visible when the super gate extraction is also implemented.

Keywords: Boolean satisfiability, static and dynamic learning, test generation, redundancy identification

[1] Currently visiting at Nara Institute of Science and Technology

1. Introduction

In recent years substantial progress has been achieved in computer aided design (CAD) for large integrated circuits using the Boolean satisfiability (SAT) method. The main reason for this progress is development and elaboration of efficient learning techniques. Two types of learning, static and dynamic, originally implemented in SOCRATES [1] find indirect implications during preprocessing as well as necessary assignments during test generation. Further improvement of the learning techniques have been achieved in ATPG systems Nemesis [2] and TRAN [3] based on the Boolean satisfiability method. The first complete learning algorithm, called *recursive learning*, has been introduced in [4]. Now, learning techniques are widely used in many CAD processes such as logic optimization, resynthesis, verification and test generation [5].

Static learning is performed as a preprocessing phase. It is a procedure that performs implications on both value assignments for all nodes of a circuit and finds static indirect implications by analysis based on learning rules. Contrary to direct implications that can be easily extracted by the structure of a circuit, the detection of indirect implications requires an analysis of the logic functions as they represent information that is not obvious from the circuit description. The goal of static learning is to examine the relation between the signals and find as many as possible static indirect implications that will be used later during branch and bound search.

Dynamic learning is performed during branch and bound search. Its goal is to find as many as possible necessary assignments (and/or dynamic indirect implications) at the current level of decision tree.

Both static and dynamic learning play a key role in avoiding unnecessary backtracking during branch and bound search by finding as many as possible necessary assignments at the current level. From a practical point of view, a vast majority of indirect implications can be easily derived during static learning, while high level recursive learning has to be performed to find them during branch and bound search. Previous work in test generation also showed that with efficient preprocessing, it is not necessary to perform dynamic learning for the vast majority of faults [6,7]. Therefore, static learning is an important preprocessing phase for test generation.

Herein we will discuss only low complexity static learning since it is the more important issue for test generation and it is easier to understand the basic concept as well as to compare different learning techniques and procedures. The paper is organized as follows. In Section 2 an SAT basis and the structure of the implication graph are provided. In Section 3 basic terminology and learning rules are examined. In Section 4 we analyze and classify learning procedures. In Section 5 we present a new data structure for the complete implication graph and give examples how it affects static and dynamic learning as well as branch and bound search. In Section 6 two efficient implicit static learning procedures are proposed. Experimental results are given in Section 7, and we conclude in Section 8.

2. SAT basis

SAT algorithms translate a problem into a characteristic formula that represents the constraints for the possible solutions. The characteristic formula is usually written in Conjunctive Normal Form (CNF) where one sum is called a clause. Clauses with one, two, three and more variables are called unary, binary, ternary and k-nary clauses respectively. For instance, $(X \vee \overline{Z}) \wedge (Y \vee \overline{Z}) \wedge (\overline{X} \vee \overline{Y} \vee Z)$ is the CNF formula of an AND gate $Z = X \wedge Y$. It consists of one ternary and two binary clauses and evaluates to 1 iff the value of variables X, Y and Z is consistent with the truth table of the corresponding logic gate.

The first step in satisfying a CNF formula is to construct an implication graph. More formally, each variable X is represented by two nodes in the implication graph labeled X and $\overline{X}$. Next, each binary clause $(X \vee Y)$ can be viewed by two *local implications* $(\overline{X} \rightarrow Y)$ and $(\overline{Y} \rightarrow X)$ between these nodes, i.e., for each binary clause $(X \vee Y)$ there are two directed edges in the graph that represent these two local implications. See Equation (1):

$$X \vee Y \Leftrightarrow (\overline{X} \rightarrow Y) \wedge (\overline{Y} \rightarrow X) \qquad (1)$$

Thus, the formula can be easily manipulated since a binding procedure requires only a partial traversal of the implication graph and a checking of the ternary clauses [2]. In some cases, global implications are also derived by static learning in order to facilitate satisfying the CNF formula.

Binding procedure:

- *Perform local and global implications:* If a node is assigned 1, then all its successors must be bound to 1.
- *Perform direct $\wedge$-implications:* If k-1 variables in a k-nary clause are specified and the clause is still unsatisfied, then the last variable must be bound to a value so that this clause evaluates to 1.
- An assignment is *inconsistent* if either (1) or (2) is true:

(1) Both nodes $\overline{X}$ and X of a variable X are marked, i.e. they have a value 1.

(2) All variables in a ternary clause are specified and the clause is still unsatisfied.

However, in this case the implication graph represents only the binary clauses. In [8], an efficient data structure that represents all clauses of the formula has been proposed. The resultant implication graph is called *complete* and contains two types of nodes. While the first type nodes represent the variables, the second type, called *$\wedge$-nodes*, symbolize the conjunction operation of two nodes or simply *direct $\wedge$-implications*. In this way, each ternary clause is uniquely represented by three direct $\wedge$-implications in the complete implication graph, see Equation (2). Equation (3) shows the

transformation of a k-nary clause into ternary clauses. Thus, the complete implication graph efficiently represents the whole formula, not only the binary clauses.

$$(X \vee Y \vee Z) \Leftrightarrow (\overline{X} \wedge \overline{Y} \rightarrow Z) \wedge (\overline{X} \wedge \overline{Z} \rightarrow Y) \wedge (\overline{Y} \wedge \overline{Z} \rightarrow X) \quad (2)$$

$$(X \vee Y \vee Z \vee W) \Leftrightarrow \exists\, V \in \{0,1\} : (X \vee Y \vee V) \wedge (Z \vee W \vee \overline{V}) \quad (3)$$

Figure 1 depicts the implications for a binary and ternary clause. This structure of the complete implication graph may be considered as independent of the chosen logic and allows parallel value assignments to be performed [9].

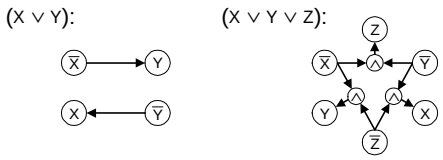
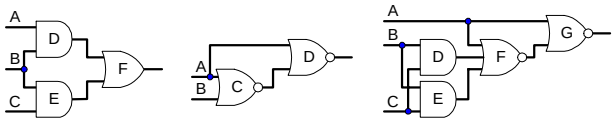


Figure 1: Implications for a binary and ternary clause

3. Basic learning rules

The precision of the premier learning procedures [1,2,3] strongly depend on the order of the value assignments because some indirect implications can be found if certain other indirect implications have already been derived. To avoid that dependency, we assume an iterative computation of the indirect implications introduced in [7], i.e. the iterative static learning procedure performs both 0 and 1 value assignments through the variables until one full iteration produces no new implications.



(a) (b) (c)

Figure 2: Network examples 1, 2 and 3

3.1. Contradiction (learning rule 1)

In [1], static learning is performed as a preprocessing phase based on contrapositive law, Equation (4), called here *a learning rule 1* where A, B∈{0,1}.

$$(X=A \rightarrow Y=B) \Leftrightarrow (Y=\overline{B} \rightarrow X=\overline{A}) \quad (4)$$

Clearly, the 2CNF portion of a formula (only the binary clauses) priory fulfills contrapositive law. This is not true if the k-nary clauses of the formula are also included. For example, it is possible that an assignment binds k-1 variables in a k-nary clause and this clause is still unsatisfied. Then a direct ∧-implication is performed and the last variable of the clause is bound to a certain value so that the clause is satisfied, i.e., it evaluates to 1. If the last variable of the clause is an output (or an input) for

the corresponding gate, then the binding procedure performs a forward (or a backward) $\wedge$-implication or a direct $\wedge$-implication (both cases). For example, assignment B=0 for the circuit in Figure 2(a), binds variables D and E to 0 and ternary clause ($D \vee E \vee \overline{F}$) corresponding to the logic gate F is still unsatisfied. Next, the binding procedure performs a forward $\wedge$-implication and sets the last variable F to 0. Thus, indirect implication ($F=1 \rightarrow B=1$) is found by contradiction.

Figure 2(b) shows an example for backward $\wedge$-implication. Assignment D=1 binds variables A and C to 0, and clause ($A \vee B \vee C$) is still unsatisfied. To satisfy this clause the binding procedure performs a backward $\wedge$-implication and sets the last variable B to 1. Thus, indirect implication ($B=0 \rightarrow D=0$) is found by contradiction.

3.2. Simplification

Equations (5) and (6) show two *auxiliary rules A.1 and A.2* known as fixation and identification of identity and exclusion, i.e. two variables assume identical or different values respectively.

$$(X=A \rightarrow Y=B) \wedge (X=\overline{A} \rightarrow Y=B) \quad (5)$$

$$(X=A \rightarrow Y=B) \wedge (Y=\overline{B} \rightarrow X=\overline{A}) \quad (6)$$

Both these rules can be used to simplify the characteristic formula by reducing some variables in the binary, ternary and k-nary clauses and in this way facilitate the derivation of new indirect implications. According to rule A.1, if Equation (5) is true then variable Y must be fixed in a value B for all value assignments. According to rule A.2, if two variables assume identical (different) values then one of them can be removed (substituted by another one anywhere in the formula). This may take an effect only in the clauses where these variables are simultaneously presented. Clearly, while the application of auxiliary rule A.1 is easy and direct, the application of auxiliary rule A.2 needs some undesirable modifications in the characteristic formula.

Figure 2(c) shows an example for auxiliary rule A.2. First, the learning procedure identifies that variables D and E are equivalent and then either variable D or E can be removed from clause ($A \vee D \vee E \vee F$). Next, the circuit in Figure 2(c) becomes similar to the circuit in Figure 2(b) and indirect implications ($D=0 \rightarrow G=0$) and ($E=0 \rightarrow G=0$) can be easily found by contradiction.

3.3. Indirect $\wedge$-implication (learning rule 2)

In [10], another concept, called an unjustified line, has been introduced. On this basis, some necessary assignments can be derived as an intersection of the conditions for satisfying a k-nary clause. This is called here *an indirect $\wedge$-implication* or *a learning rule 2*.

The learning procedure based on rule 2 finds indirect implication ($L=1 \rightarrow D=1$) in the circuit shown in Figure 3(a). More precisely, assignment L=1 binds variables K, J and F to 0 and k-nary clause ($F \vee G \vee H \vee I \vee J \vee K$) is still unsatisfied. To satisfy this clause either variable G, H or I must be set to 1, but all these assignments imply that variable D should be bound to 1. Therefore D=1 is a necessary

assignment since it is an intersection of the conditions for satisfying unjustified line K. Thus, indirect implications (L=1 → D=1) and (D=0 → L=0) are found by contradiction. Clearly, these indirect implications cannot be found by the previous learning rules.

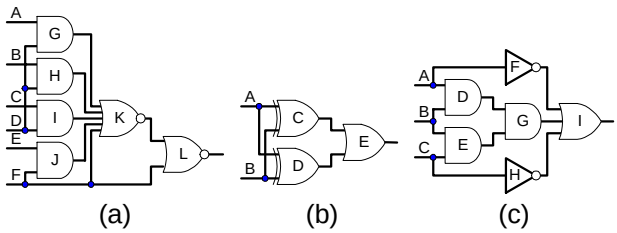


Figure 3: Network examples 4, 5 and 6

3.4. Hierarchy

A hierarchy can be used to find some indirect implications that cannot be found by the previous rules. Figure 3(b) depicts an example where the learning procedure cannot identify equivalence between variables C, D and E, i.e. these variables assume an identical value. After applying *an auxiliary rule A.3*, decomposition of all XOR gates into basic gates, the iterative learning procedure finds the equivalence between these variables by contradiction. Figure 3(c) shows an example for super gates extraction called here *an auxiliary rule A.4*. In this circuit, the gates D, E and G form a super gate [11]. If we replace these three gates by their super gate (3-input AND gate), then the indirect implications (I=0 → B=0) and (B=1 → I=1) can be found by contradiction. In the original circuit, these indirect implications cannot be found by applying the previous learning rules.

Unfortunately, application of rules A.3 and A.4 needs some undesirable modifications in the formula. More precisely, spare variables have to be added (for rule A.3) and some gates of the circuit can be included in more than one super gate (for rule A.4).

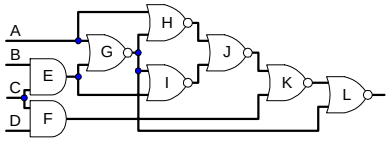


Figure 4: Network example 7 [12]

3.5. Recursive learning (learning rule 3.N)

The first complete learning algorithm, called *recursive learning*, has been introduced [4]. If the level of recursion N is not restricted, this algorithm can find all static indirect implications during static learning and all necessary assignments during branch and bound search. For example, the previous learning rules cannot find static indirect implication (L=1 → C=1) in the circuit shown in Figure 4, while this is possible by applying a first level recursive learning, called here *a learning rule 3.N* when N=1. During iterative value assignments through the variables, the static learning procedure first finds indirect implication (E=0 → H=0) by contradiction since assignment H=1 binds variable E to 1. Next,

assignment C=0 binds variables E, F and H to 0, and ternary clause (A$\vee$ G $\vee$ H) is still unsatisfied. To satisfy this ternary clause, either variable A or G must be set to 1, but both these assignments bind variable L to 0. Therefore L=0 is a necessary assignment and indirect implications (C=0 $\rightarrow$ L=0) and (L=1 $\rightarrow$ C=1) are found by contradiction. These indirect implications cannot be found by the previous learning rules since neither assignment A=1 nor G=1 priory implies L=0.

Observation 1: For iterative static learning, rule 1 covers rule A.1.

Observation 2: For iterative static learning, rule 2 covers rules A.2 and 1.

Observation 3: For iterative static learning, rule 3.1 covers rules A.3 and 2.

Observation 4: For iterative static learning, rule 3.N covers rule A.4 only when the parameter N is not restricted, see Example 1.

Corollary 1: Learning rules 1 (contradiction), 2 (indirect $\wedge$-implication) and 3.N (recursive learning) can be considered as basic learning rules.

Corollary 2: Auxiliary rules A.1, A.3 and A.4 are useful for decreasing the complexity and/or increasing the precision of the static learning procedures because of the higher complexity of the covering learning rules 1, 3.1 and 3.N respectively.

Example 1: Let us consider an example for Observation 4. As a result of performing the iterative static leaning based on rule 3.1 for the iredundant circuit shown in Figure 5(a), neither indirect implication nor indirect $\wedge$-implication is found. To find indirect implication (R=1 $\rightarrow$D=1) for this circuit, the iterative learning procedure must apply rule 3.2 (second level recursive learning), while this indirect implication can be easily found using rules 2 and A.4 (super gate extraction). The modified circuit by auxiliary rule A.4 is shown in Figure 5(b).

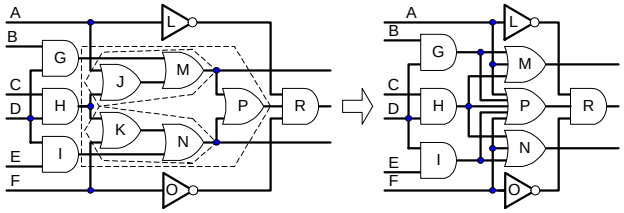


Figure 5: Network example 8 before and after applying rule A.4 (super gate extraction)

4. Overview of learning procedures

In this section, we analyze and classify well-known learning procedures by eight network examples presented in the previous section. Because of the iterative nature of static learning, the newly derived implications are difficult to identify. To do so, the learning procedures use a structure, clause, implication graph and set algebra based methods.

4.1. Structure based method

The SOCRATES static learning procedure [1] temporarily sets a variable to a value 0 or 1 and computes all implications deduced by this assignment. Then it checks if each implied value is non-controlling for the corresponding gate output (learning criterion). In this way, the SOCRATES static learning procedure identifies the indirect implications as a result of performing only the forward $\wedge$-implications. It cannot identify the indirect implication (B=0 $\rightarrow$ D=0) for the circuit in Figure 2(b) since assignment D=1 binds variable B to 1 by performing a backward $\wedge$-implication and variable B does not correspond to any gate output.

4.2. Clause based method

In [2], an improved static learning procedure based on rule 1 is presented. The binding procedure (see Section 2) has two phases: first it performs all local and global implications and then all direct $\wedge$-implications. Thus, newly derived indirect implications can be easily identified by contradiction since they must involve at least one direct $\wedge$-implication (reduction of a ternary or k-nary clause to unary). A new indirect implication is included in the formula by adding a new binary clause (two global implications in the implication graph). In this way, Nemesis and TEGUS find an indirect implication (B=0 $\rightarrow$ D=0) for the circuit shown in Figure 2(b). First, assignment D=1 binds variables A and C to 0. Since ternary clause (A $\vee$ B $\vee$ C) is unsatisfied, the binding procedure performs a direct $\wedge$-implication and the last variable B is bound to 1. In this case, the necessity of the performing a direct $\wedge$-implication means that implication (D=1 $\rightarrow$ B=1) is not available in the implication graph and the implication (B=0 $\rightarrow$ D=0), derived by contradiction, is indirect. Next, two global implications, (B=0 $\rightarrow$ D=0) and (D=1 $\rightarrow$ E=1), representing the clause ($\overline{\text{D}}$ $\vee$ B) are added in the implication graph. The SOCRATES learning procedure cannot find this static indirect implication.

4.3. Implication graph based method

TRAN [3] learning procedure derives indirect implications by finding a transitive closure of an implication graph and checking for certain properties on it. TRAN uses fixation, identification and exclusion to simplify the formula, i.e. to transform the k-nary clauses to binary clauses in order to include them in the implication graph. TRAN also uses contradiction and fixation to derive indirect implications. Fixation is equivalent to rule 3.1 which is restricted to the unsatisfied clauses with two unspecified variables since this learning procedure makes an analysis only on the implication graph. This is why TRAN can find some difficult indirect implications, such as (C=0 $\rightarrow$ L=0) and (L=1 $\rightarrow$ C=1) in network example 7, but fails to find easier indirect implications, such as (L=1 $\rightarrow$ D=1) and (D=0 $\rightarrow$ L=0) in network example 4. On the other hand, the dynamic update of the implication graph and the calculation of the transitive closure make this procedure complicated and costly.

4.4. Set algebra based method

In [12], an iterative learning procedure based on set algebra and learning rule 3.1 is presented. It calculates two sets, called here *transitive* and *contrapositive*, for each signal value assignment. The transitive *set* for an assignment is a list of all implications for this assignment, while the contrapositive *set* for an assignment is a list of all assignments that imply this assignment by contradiction. For example, if assignment A=0 binds variable B to 1, then implication B=1 is included in the transitive set of assignment A=0 and implication A=1 is included in the contrapositive set of assignment B=0.

Table 1 provides the results of our analysis of the precision and complexity of the well-known learning procedures. The precision is evaluated by the learning rules and network examples presented in the previous section. In respect to their complexity, the learning procedures are divided into three categories: *low*, O(MN), *average*, O(M^2N), and *high*, O(M^3N), where N is the number of the nodes and M is the average number of the nodes set by every signal value assignment. We can see that network example 8 is beyond the scope of most of the learning procedures because of the high complexity of recursive learning [4]. Of course, there are exceptions [9] where static learning based on rule 3.3 (third level recursive learning) is used for logic verification.

Table 1: The well-known learning procedures

Learning procedures	Network examples								Precision	Complexity
	1	2	3	4	5	6	7	8		
SOCRATES [1]	+	-	-	-	-	-	-	-	< rule 1	Low
Nemesis [2]	+	+	+	-	-	-	-	-	> rule 1	Low
TRAN [3]	+	+	+	-	+	+	+	-	< rule 3.1	High
TEGUS [7]	+	+	-	-	+	-	-	-	> rule 1	Low
Simprid [12]	+	+	+	+	+	+	+	-	rule 3.1	Average

5. Proposed data structure of the complete implication graph

In this section, we present an alternative solution for the complete implication graph, and show how this new data structure can be used to facilitate the derivation and performance of the indirect $\wedge$-implications. Finally, we give three examples how this approach affects static learning, branch and bound search, and dynamic learning.

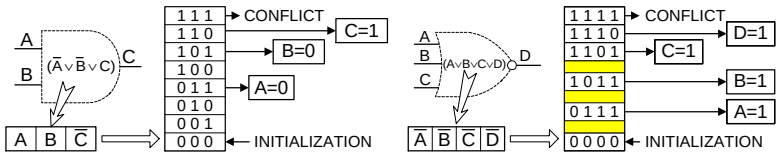


Figure 6: The representation of k-nary clauses

The new data structure of the complete implication graph represents each gate (k-nary clause) by 2^k $\wedge$-nodes, k direct $\wedge$-implications and one k-bit key dynamically calculated by the binding procedure, see Figure 6. For the basic gates, each bit of a k-bit key corresponds to one of the variables in a k-nary clause. The bit is set to one if this variable is specified and the clause is still unsatisfied. The k-bit key also has one extra bit, not presented in Figure 6, set to 1 when the clause is satisfied. To

represent the 2-input XOR gate, this data structure needs a 6-bits key that is an ordered set of the six nodes corresponding to the variables of the 2-input XOR gate. In this case, certain patterns will indicate the conflict situations when all variables are specified and one of the corresponding three clauses is still unsatisfied. This data structure of the complete implication graph allows a unified representation of both the ternary and k-nary clauses and makes them easier for manipulation. It also allows both static and dynamic indirect implications based on rule 2 to be easily found.

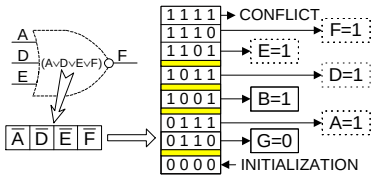


Figure 7: The representation of indirect $\wedge$-implications

Example 2: Let us consider how indirect implication (G=1 $\rightarrow$ B=1) in the circuit shown in Figure 2(c) can be easily found during static learning. First, assignment B=0 binds variables D and E to 0 and k-nary clause (A $\vee$ D $\vee$ E $\vee$ F) is still unsatisfied. To take into account this relation, the learning procedure adds static indirect $\wedge$-implication, implication B=1 to $\wedge$-node <1001> of gate F, see Figure 7. Next, assignment G=1 binds variables A and F to 0 and the k-nary clause (A $\vee$ D $\vee$ E $\vee$ F) is still unsatisfied. To take into account this relation, the learning procedure adds static indirect $\wedge$-implication, implication G=0 to $\wedge$-node <0110> of gate F, see Figure 7. Clearly, the superposition of these two keys <1001> and <0110> will result in <1111> therefore, these two assignments are inconsistent, i.e., if both assignments are preformed then all variables of the k-nary clause (A $\vee$ D $\vee$ E $\vee$ F) will be specified and this clause will be still unsatisfied. In this way, the static learning procedure easily finds (without spare operations for calculation of an intersection of the transitive sets), indirect implications (G=1 $\rightarrow$ B=1). Thus, the proposed data structure of the complete implication graph allows the complexity of static learning based on rule 2 to be considerably decreased. In a similar way, the indirect implications (L=1 $\rightarrow$ D=1) and (D=0 $\rightarrow$ L=0) in the circuit shown in Figure 3(a) can be found.

Lemma 1: A static indirect $\wedge$-implication is *effective*, if the corresponding clause has at least four variables, i.e. it must be a k-nary clause.

Proof: To prove Lemma 1, it is sufficient to show that any static indirect $\wedge$-implication of a ternary clause cannot produce a new indirect implication.

(case A) Let an assignment set two variables in a ternary clause and the clause be unsatisfied. In this case, direct $\wedge$-implication has to be performed, i.e. the indirect $\wedge$-implication will be ineffective.

(case B) Let an assignment set only one variable in a ternary clause and the clause be unsatisfied. In this case, if an indirect implication exists, then it can be found by contradiction, i.e., the indirect $\wedge$-implication will be ineffective. *(end of proof)*

Corollary 3: To produce an effective indirect $\wedge$-implication, a value assignment has to bind at least two variables in a k-nary clause.

Lemma 2: Let us consider all indirect $\wedge$-implications ($\underline{Si}$=<si1,si2,..,sik> $\rightarrow$ Xi=Ai) of a k-nary clause where Xi=Ai is an assignment and $\underline{Si}$ is an $\wedge$-node for this clause. Then an indirect $\wedge$-implication ($\underline{Si} \rightarrow$ Xi=Ai) is *ineffective* iff there exists another indirect $\wedge$-implication ($\underline{Sj} \rightarrow$ Xj=Aj) such that ($\underline{Sj}$ = $\underline{Si}$) $\wedge$ (Xj=Aj $\rightarrow$ Xi=Ai).

Proof: Both necessity and sufficiency have to be examined to prove Lemma 2.

(necessity) Let $\underline{Si}$=$\underline{Sj}$, then implication (Xj=Aj $\rightarrow$ Xi=Ai) guarantees by transitive law that assignment Xj=Aj binds all variables set by assignment Xi=Ai. Therefore, the indirect $\wedge$-implication ($\underline{Si} \rightarrow$ Xi=Ai) is ineffective.

(sufficiency) Let indirect $\wedge$-implication ($\underline{Si} \rightarrow$ Xi=Ai) be ineffective. Then, another indirect $\wedge$-implication ($\underline{Sj} \rightarrow$ Xj=Aj) must exist such that $\underline{Sj} \subseteq \underline{Si}$ since all inconsistent assignments for assignment Xi= $\overline{Ai}$ must also be inconsistent for assignment Xj= $\overline{Aj}$. Let $\underline{Sj} \subset \underline{Si}$, i.e. assignment Xj= $\overline{Aj}$, set more variables in a k-nary clause than assignment Xi= $\overline{Ai}$. Then, the assignment Xj=Aj cannot set variable Xi to value Ai. Hence, the indirect $\wedge$-implication ($\underline{Si} \rightarrow$ Xi=Ai) is effective.

(end of proof)

Corollary 4: To be an effective indirect $\wedge$-implication ($\underline{S} \rightarrow$ X=A), all the variables corresponding to the previous implications for $\wedge$-node $\underline{S}$ must be unspecified for the assignment X= $\overline{A}$.

Property 1: The number and importance of the indirect $\wedge$-implications tends to increase when the number and size of the k-nary clauses increases.

Property 2: Without postprocessing reduction, the number of the indirect $\wedge$-implications found during iterative static learning will depend on the order in which the assignments are performed.

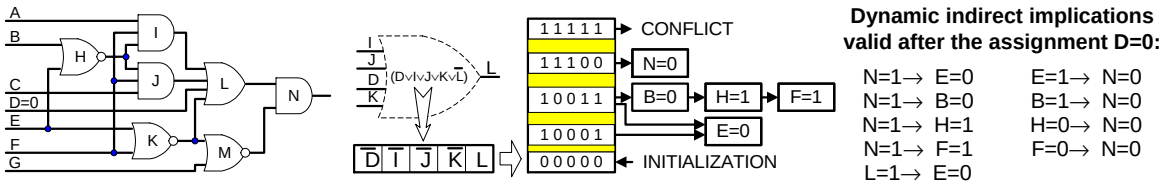
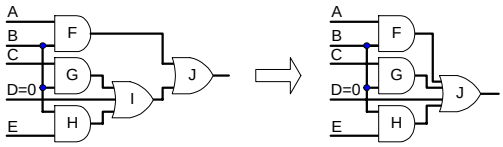


Figure 8: Network example 9

Example 3: Figure 8(a) provides an example of the transformation of indirect $\wedge$-implications into dynamic indirect implications during branch and bound search. For this circuit, 0 indirect implications and 5 indirect $\wedge$-implications are found by static learning. They are depicted in Figure 8(b). This example considers two specific cases that the learning procedure has to take into account when it derives and performs the static indirect $\wedge$-implications.

(case 1) According to Lemma 2, implication H=1 makes implication B=0 ineffective since both implications belong to one $\wedge$-node and assignment H=1 binds variable B to 0 by local implication (H=1 $\rightarrow$ B=0). The static learning procedure can avoid this, if the assignment H=0 is performed before assignment B=1 and Corollary 4 is applied.

(case 2) After assignment D=0 during branch and bound search, 9 dynamic indirect implications, shown in Figure 8(c), become valid. This effect can be achieved without any spare operations, if the indirect $\wedge$-implications are correctly performed. For example, when the k-bit key of gate L is <10011> then indirect $\wedge$-implication (<10001> $\rightarrow$ E=0) is effective, i.e. it must be performed, otherwise dynamic indirect implication (N=1 $\rightarrow$ E=0) will be discarded.



(a) (b)

Figure 9: Network example 10 before and after the super gate extraction

Example 4: Figure 9 provides an example of how the indirect $\wedge$-implications and super gate extraction can affect dynamic learning and branch and bound search. As a result of the static learning based on rule 2, only one indirect $\wedge$-implication is found for the iredundant circuit shown in Figure 9(a). After assignment D=0, this indirect $\wedge$-implication becomes a valid dynamic indirect implication (I=1 $\rightarrow$ B=1). Using this dynamic indirect implication, implication (J=1$\rightarrow$ B=1) can be found by dynamic learning based on rule 3.1 (first level recursive learning), otherwise dynamic learning based on rule 3.2 (second level recursive learning) must be applied. This implication can be found without dynamic learning if the super gate extraction is applied before static learning. In this case, gates I and J form super gate J, see Figure 9(b) and the dynamic indirect implication (J=1 $\rightarrow$ B=1) become valid after the assignment D=0.

6. Proposed static learning procedures

We developed two Implicit Static Learning Procedures, ISLP1 and ISLP2, using the new data structure for the complete implication graph. The first procedure is clause based, while the second is mixed clause and set algebra based. Before static learning the complete implication graph is built, and all local implications and direct $\wedge$-implications are extracted from the circuit description. The binding procedure of ISLP1, like Nemesis, first performs all implications (local and global) and then all $\wedge$-implications (direct and indirect), while the binding procedure of ISLP2 between the implications and the $\wedge$-implications performs the implications using the iterative calculated contrapositive sets.

Both procedures consider a stem and its fanout branches as one node. Initially, the set of variables includes all nodes without the nodes corresponding to the outputs of EQU and NOT gates since they are equivalent and antivalent to their inputs. During static learning, if two variables are identified as

equivalent or antivalent, one of them is removed from the set of variables. Also, if the learning procedure finds a variable that needs a constant value, it is permanently fixed to this value.

To decrease the number of assignments through the variables, ISLP1 and ISLP2 use specific approaches. During the first pass through the variables, both value assignments 0 and 1 are performed. During the next passes, the learning procedures perform an assignment only if it can produce a new indirect implication. For example, ISLP2 performs an assignment if at least one new indirect implication has been added to the corresponding contrapositive set. Although this criterion does not guarantee completeness in respect to learning rule 2, all benchmark circuits have been processed successfully and all indirect implications have been derived.

7. Experimental results

We implemented ISLP1 and ISLP2 in C programming language and ran experiments on a 450MHz Pentium-III PC (SpecInt'95=18.7). Table 2 shows the results: the number of nodes, variables, constant value assignments and direct implications before static learning and the numbers of reduced variables, constant value assignments, indirect implications and indirect $\wedge$-implications after static learning based on rule 1 and rule 2. The contribution of rule 1 and rule 2 is 2049859 and 3888 static indirect implications. We counted the implications as presented in [12]: (1) if a node sets itself and X other nodes then X+1 implications were counted; and (2) the constant value assignments were not counted as implications after static learning. Therefore the correct number of the implications before and after static learning, IB and IA, can be calculated by Equations (7) and (8):

$$IB=DI - 2N + CB \quad (7)$$

$$IA=DI + INDI + (2N - CA)(CA - 1) \quad (8)$$

where N is the number of the nodes, CB and CA are the numbers of the constant value assignments before and after static learning, and DI and INDI are the numbers of the direct and indirect implications. In addition, 235630 static indirect $\wedge$-implications were derived by the static learning procedure based on rule 2. We counted an indirect $\wedge$-implication if a value assignment set at least two variables in a k-nary clause (where $k \geq 4$).

Table 3 shows the distribution of the implications before and after static learning based on rule 2. It may appear confusing that the maximal numbers of implications for the circuits S15850 and S38584 after static learning are smaller than before static learning. The reason is that the constant value assignments were not counted as implications after static learning. Clearly, if the static learning procedure finds 168 constant value assignments for circuit S38584, then after static learning each assignment sets at least 168 nodes.

Table 4 provides a comparison of the experimental results of the proposed static learning procedures with those of Simprid [12]. To do that, we changed our procedures so that the number of nodes became identical to that reported in [12], i.e., #node = #gates + #primary inputs + #primary outputs. In this way, the nodes corresponding to the primary outputs were doubled and some

implications were counted twice. Columns (3) and (4) give the number of constant value assignments and the number of static implications for the learning procedures based on rule 2. Columns (8) and (9) give the differences in these two parameters for the learning procedure based on rule 3.1. All static learning procedures found identical numbers of constant value assignments, while the static learning procedure based on rule 3.1 derived 3040 more hard-to-detect indirect implications for the ISCAS'85[13] benchmark circuits. Columns (5) and (6) give the CPU time in seconds for ISLP1 and ISLP2, while those parameters measured by HP 9000 J200 (SpecInt'95=4.98) are given in columns (7) and (10). The proposed static learning procedures were about 20 times more efficient because of the new data structure of the complete implication graph that facilitates the performing of $\wedge$-implications and avoids costly calculation of the intersection of the transitive sets when applying learning rule 2.

To evaluate the impact of the new data structure on redundancy identification, we used the combinational ATPG system SPIRIT[15] based on the Boolean satisfiability method and the Single Path Oriented Propagation (SPOP) method [6]. Since SPIRIT processes a single output circuit, it is necessary to prove that a fault is redundant in respect to each primary output where the fault effect can be observed. Therefore, more than one test session for some faults had to be performed. As a criterion we chose four parameters: (P1) the number of test sessions, (P2) the maximal number of value assignments within one test session, (P3) the number of sensitized unjustifiable propagation paths and (P4) the number of aborted test sessions/faults. Table 5 shows the results in three cases: (case 1) without static learning, (case 2) static learning based on rule 1 and (case 3) static learning based on rule 2. For learning rule 1 (learning rule 2), the maximal number of assignments was 449 (279) and the number of sensitized unjustifiable propagation paths was 34 (32) respectively. Therefore for these parameters, the contribution of rule 2 was 37.8% and 5.9% respectively.

Let us assume that the preprocessing time is negligible in respect to the time spend for branch and bound search. Since we did not apply dynamic learning and search pruning techniques in this experiment then the number of assignments (the size of decision tree) can be associated with the complexity for redundancy identification. Therefore with the above mention restrictions, the maximal number of assignments (P2) gives an idea about impact of the precision of static learning on the worst-case complexity for the redundancy identification. We observed two cases where a reduction of 170 and 2 times in parameter P2 was achieved after applying rule 2, for the circuit C432 and C1908. We also observed two cases where the number of the sensitized unjustifiable propagation paths were decreased after applying learning rule 2, for the circuit C432. In all these cases, the effect was from the performing the static indirect $\wedge$-implications during SPOP.

8. Conclusions

We have presented a new type of indirect implication and a new data structure for the complete implication graph that decreases the complexity of learning based on rule 2. We showed that by deriving and performing the indirect $\wedge$-implications, some additional static and dynamic indirect implications are easily found which affect both static and dynamic learning as well as branch and bound search. On this basis, two efficient implicit static learning procedures have been proposed. Experimental results confirmed their efficiency and precision. Using the new data structure of complete implication graph, 3888 more hard-to-detect static indirect implications as well as 235630 static indirect $\wedge$-implications were derived keeping the complexity of static learning as low as possible. As a result, the maximal number of value assignments within one test session and the number of sensitized unjustifiable propagation paths for the redundant faults in the ISCAS'85 and ISCAS'89 benchmark circuits during SPOP has been reduced by 37.8% and 5.9% respectively. On the other hand, a quick look at the results in [11] shows that the ratio between the number of gates and super gates for the benchmark circuits varies from 1.01 (for the circuit C6288) to 4.46 (for the circuit S641), while the average ratio is about 2.1. Taking into account this fact as well as Corollary 2, Property 1, and Examples 1 and 4, we expect that the contribution of the new data structure will be more visible after implementation of the super gate extraction (auxiliary rule A.4).

Acknowledgment

This paper was supported by JSPS under grant P99747.

References:

1. M.Schulz, E.Trischler and T.Sarfert, "SOCRATES: A High Efficient Automatic Test Pattern Generation System," IEEE Trans. on CAD, vol. 7, No.1, Jan. 1988, pp.126-137.

2. T.Larrabee, "Test Pattern Generation using Boolean Satisfiability," IEEE Trans. on CAD, vol.11, No.1, Jan.1992, pp.4-15.

3. S.Chakradhar, V.D.Agarval and S.Rothweiler, “A Transitive Closure Algorithm for Test Generation,” IEEE Trans. on CAD, vol. 12, No.7, July 1993, pp.1015-1028.

4. W.Kunz and D.K.Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," Proc. IEEE ITC, 1992, pp.816-825.

5. W.Kunz and D.Stoffel, "Reasoning in Boolean Networks," Kluwer Academic Publisher, 1997, pp.230.

6. M.Henftling, H.Wittmann and K.Antreich, "A Single-Path-Oriented Fault-Effect Propagation in Digital Circuits Considering Multiple-Path Sensitization," Proc. IEEE/ACM ICCAD, 1995, pp.304-309.

7. P.Stephan, R.K.Brayton and A.L.Sangiovanni-Vincentelli, ”Combinational Test Generation using Stisfiability,” IEEE Trans. on CAD, vol. 15, No.9, Sept. 1996, pp.1167-1176.

8. M.Henftling and H.Wittmann, “A New Data Structure to Solve the Satisfiability Problem in Digital Circuits,” Archiv für Elektronik und Übertragungstechnik, January 1995, pp.29-43.

9. P.Tafertshofer, A.Ganz and M.Henftling, "A SAT-Based Implication Engine," Technical University of Munich, Technical Report TUM-LRE-97-2, pp.1-26.

10. H.Fujiwara and T.Shimono, ”On Acceleration of Test Generation Algorithms,” IEEE Trans. on Computers, vol. C-33, December 1983, pp.1137-1144.

11. K.H.Tsai, R.Tompson, J.Rajski, M.Marek-Sadowska, “STAR-ATPG: A High Speed Test Pattern Generator for Large Scan Designs,” Proc. IEEE ITC, 1999, pp.1021-1030.

12. J.Zhao, E.Rudnick and J.Patel, "Static Logic Implication with Application to Redundancy Identification," Proc. IEEE VLSI Test Symposium, 1997, pp. 288-293.

13. F.Brglez and H.Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and A Target Translator in Fortran," Proc. IEEE ISCAS, 1985, pp. 663-698.

14. F.Brglez, D.Bryan and K.Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," Proc. IEEE ISCAS, 1989, pp.1929-1934.

15. E.Gizdarski, "Test Pattern Generation using Boolean Satisfiability," Automatica and Informatics, vol-32, No.2, 1998, pp. 33-40.

Table 2: Static learning results

Circuit	#nodes	before learning			Learning rule 1			Learning rule 2			
		#var	#const	#direct implications	#red. var.	#const	#indirect implications	#red. var.	#const	#indirect implic.	#indirect $\wedge$-implic.
C432	196	156	0	2218	0	0	236	+0	+0	+4	+137
C499	243	203	0	6526	0	0	648	+0	+0	+0	+48
C880	443	354	0	5957	7	0	261	+0	+0	+0	+468
C1355	587	515	0	27574	0	0	3904	+0	+0	+0	+64
C1908	913	474	0	37777	116	0	8935	+0	+0	+0	+1201
C2670	1426	909	3	50459	137	10	7577	+0	+0	+0	+2292
C3540	1719	1006	1	272849	185	1	38511	+0	+0	+0	+23972
C5315	2485	1591	1	75721	192	1	23561	+1	+0	+954	+10301
C6288	2448	2416	17	20977	33	17	8633	+0	+0	+0	+0
C7552	3719	2309	2	182713	367	4	113645	+0	+0	+68	+10567
S1238	540	460	0	44819	2	0	16529	+0	+0	+180	+3488
S5378	2993	1218	25	175590	21	25	37610	+0	+0	+0	+9238
S9234	5844	2274	14	723350	322	14	237752	+0	+0	+320	+16931
S13207	8651	3273	23	1116646	236	23	262618	+0	+0	+106	+36509
S15850	10383	4059	34	1401941	262	34	255081	+0	+0	+1186	+46498
S35932	17828	13967	0	9124712	544	0	451216	+0	+0	+0	+0
S38417	23843	10373	6	1516297	529	6	201649	+0	+0	+0	+10290
S38584	20717	12912	160	20622207	648	168	381493	+6	+0	+1070	+63626
Total:	**104978**	**58469**	**286**	**35408333**	**3601**	**303**	**2049859**	**+7**	**+0**	**+3888**	**+235630**

Table 3: Distribution of the implications before and after static learning

Circuit	Before learning						Learning rule 2					
	0	<10	<100	<1000	≥1000	max	0	<10	<100	<1000	≥1000	max
C432	0	299	93	0	0	31	0	289	103	0	0	31
C499	0	340	146	0	0	98	0	338	148	0	0	98
C880	0	666	220	0	0	68	0	649	237	0	0	68
C1355	0	827	235	112	0	180	0	737	325	112	0	183
C1908	0	851	876	99	0	194	0	656	1057	113	0	218
C2670	3	1804	959	86	0	260	20	1496	1240	96	0	264
C3540	1	1614	1138	685	0	705	2	1136	1424	876	0	706
C5315	1	2809	2126	34	0	200	2	2242	2626	100	0	368
C6288	17	4798	79	2	0	129	34	3588	1271	3	0	153
C7552	2	4309	2623	504	0	400	8	3030	3632	762	6	1549
S1238	0	595	307	178	0	358	0	437	388	255	0	402
S5378	25	2603	2859	499	0	336	50	2252	3045	639	0	393
S9234	14	4441	5547	1684	2	1018	28	3243	6193	2217	7	1060
S13207	23	8177	7668	1072	362	1203	46	7397	7961	1524	374	1260
S15850	34	9321	7133	4133	145	1564	68	8028	7404	5102	164	1561
S35932	0	23905	6076	1658	4017	2084	0	23869	6112	1658	4017	2086
S38417	6	29060	14756	3864	0	571	12	25433	17899	4342	0	587
S38584	160	20274	11507	2710	6783	3594	336	18321	12425	3181	7171	3491
Total:	**286**	**116693**	**64348**	**17320**	**11309**	**3594**	**606**	**103141**	**73490**	**20980**	**11739**	**3491**

Table 4: Comparison of the static learning results

Circuit	#nodes	Static learning based on rule 2					Learning based on rule 3.1		
		#const	#implications	CPU1 time, s	CPU2 time, s	CPU[12] time,s	#const	#implications	CPU[12] time,s
1	2	3	4	5	6	7	8	9	10
C432	203	0	2734	0.03	0.03	0.4	+0	+72	0.6
C499	275	0	7366	0.03	0.04	1.4	+0	+0	1.4
C880	469	0	6990	0.04	0.04	0.5	+0	+16	0.6
C1355	619	0	31990	0.11	0.15	2.9	+0	+0	3.6
C1908	938	0	47290	0.19	0.18	4.8	+0	+150	6.1
C2670	1566	11	61646	0.20	0.21	12.3	+0	+12	15.5
C3540	1741	1	313192	0.85	0.86	77.5	+0	+278	92.7
C5315	2608	1	107046	0.45	0.49	25.0	+0	+1084	51.4
C6288	2480	17	30024	0.37	0.45	11.0	+0	+972	49.1
C7552	3827	4	301608	1.41	1.26	131.3	+0	+456	182.2
S1238	572	0	65278	0.31	0.42	-	-	-	-
S5378	3221	29	239542	0.51	0.60	-	-	-	-
S9234	6094	16	996266	1.74	1.76	-	-	-	-
S13207	9441	25	1493176	5.38	5.53	-	-	-	-
S15850	11067	38	1765132	2.94	3.37	-	-	-	-
S35932	19876	0	9664336	32.01	53.25	-	-	-	-
S38417	25585	6	1824540	3.32	4.35	-	-	-	-
S38584	22447	198	21790876	33.47	60.02	-	-	-	-
Total:	**113029**	**346**	**38749032**	**83.36**	**133.01**	**267.1**	**+0**	**+3040**	**403.2**
Normalized CPU time for ISCAS'85				**68.8**	**69.4**	**1330.2**			**2007.9**
Normalized CPU time for ISCAS'89				**1558.8**	**2484.3**				

Table 5: Redundancy identification results

Circuit	Faults	Without static learning				After static learning using rule 1 or 2					
		P1	P2	P3	P4	P1	Rule 1		Rule 2		P4
							P2	P3	P2	P3	
C432	4	4	1899	3	1/1	4	170	2	**1**	**0**	0/0
C499	8	128	1	0	0/0	128	1	0	1	0	0/0
C1355	8	128	1	0	0/0	128	1	0	1	0	0/0
C1908	9	9	126	2	0/0	9	2	0	**1**	0	0/0
C2670	117	389	4769	22	12/11	357	115	9	115	9	0/0
C3540	137	655	64	14	0/0	654	34	0	34	0	0/0
C5315	59	224	4	0	0/0	222	4	0	4	0	0/0
C6288	34	754	3	0	0/0	377	1	0	1	0	0/0
C7552	131	571	690	13	0/0	562	11	0	11	0	0/0
S1238	69	185	24	20	0/0	185	17	7	17	7	0/0
S5378	40	218	6	0	0/0	205	2	0	2	0	0/0
S9234	452	747	602	227	0/0	745	20	0	20	0	0/0
S13207	151	1022	51	16	0/0	1016	50	16	50	16	0/0
S15850	389	11249	12	0	0/0	10826	12	0	12	0	0/0
S35932	3984	8704	3	0	0/0	8704	3	0	3	0	0/0
S38417	165	726	52	8	0/0	689	2	0	2	0	0/0
S38584	1506	1967	31	128	0/0	944	4	0	4	0	0/0
Total:	**7263**	**27680**	**8338**	**453**	**13/12**	**25755**	**449**	**34**	**279**	**32**	**0/0**