A SAT-Based Implication Engine

Paul Tafertshofer, Andreas Ganz, and Manfred Henftling

Technischer Bericht Nr.: TUM-LRE-97-2
22. April 1997

Technische Universität München
Lehrstuhl für Rechnergestütztes Entwerfen

A SAT-Based Implication Engine

†Paul Tafertshofer †Andreas Ganz ‡Manfred Henftling

†Institute of Electronic Design Automation
Technical University of Munich
80290 Munich,Germany

{tafertshofer,ganz}@regent.e-technik.tu-muenchen.de

‡Siemens AG, HL IT PE 5
P.O.-Box 801709
81617 Munich, Germany

mah@earl.hl.siemens.de

Abstract

This paper presents a flexible and efficient approach to evaluating implications as well as deriving indirect implications in logic circuits. Evaluation and derivation of implications are essential in ATPG, equivalence checking, and netlist optimization. Contrary to other methods, our approach is based on a graph model of a circuit's clause description called the implication graph. It combines both the flexibility of SAT-based techniques and high efficiency of structure based methods. As the proposed algorithms operate only on the implication graph, they are independent of the chosen logic. Evaluation of implications and computation of indirect implications are performed by simple and efficient graph algorithms. Experimental results for various applications relying on implication demonstrate the efficiency of our approach.

1 Introduction

Recently, substantial progress has been achieved in the fields of Boolean equivalence checking and optimization of netlists. Techniques for deriving indirect implications, which were originally developed for ATPG tools, play a key role in this development.

Indirect implications have been successfully applied in algorithms for optimizing netlists. For this task, either a set of permissible transformations is derived [1, 2, 3] or promising transformations are applied and their permissibility is later verified by an ATPG tool [4, 5, 6]. Furthermore, they are of great importance in ATPG-based approaches to Boolean equivalence checking of both combinational and sequential circuits [7, 8, 9, 10, 11, 12] as they help identify equivalent internal signals in the circuits to be compared.

In the late 1980s, Schulz et al. incorporated computation of indirect implications into the ATPG tool SOCRATES[13]. Indirect implications are indispensable when dealing with redundant faults as they help to efficiently prune the search space of the branch-and-bound search. In order to derive more indirect implications, the originally static technique of SOCRATES, which the authors refer to as *learning*, has been extended to dynamic learning [14, 15].

Recursive learning [7], proposed by Kunz et al. in 1992, was the first complete algorithm for determining indirect implications. As the problem of finding all indirect implications is NP-complete, only small depths of recursion are feasible. Recently, it has been shown that recursive learning can be adequately modelled by AND-OR reasoning graphs [3]. Another complete method for deriving indirect implications based on BDDs was suggested by Mukherjee et al. [16]. Its complexity is related to the size of the required BDDs which may grow exponentially.

Contrary to the above methods, which work on the structural description of a circuit, other approaches use a Boolean satisfiability (SAT) based model. The SAT-model allows an elegant problem formulation which can easily be adapted to various logics. This abstraction, however, often impedes development of efficient algorithms as structural information is lost. Larrabee included a clause based formulation of Schulz's algorithm into NEMESIS[17]. Her approach has been improved by the iterated method of TEGUS [18]. The transitive closure algorithms suggested by Chakradhar et al. rely on a relational model of binary clauses [19]. Silva et al. proposed another form of dynamic learning in GRASP [20] where indirect implications are determined by a conflict analysis during the backtracking phase of a SAT-solver.

In many areas of logic synthesis and formal verification Binary Decision Diagrams (BDD) have become the most widely used data structure as they provide many advantageous properties, e.g. canonicity and high flexibility. Besides their exponential memory complexity, when used for ATPG, equivalence checking, and optimization of large netlists, BDDs suffer from the drawback that implications cannot be derived efficiently on this data structure. For a given signal assignment it can only be decided if another signal assignment is implied or not. So, finding all possible implications from a given signal assignment is expensive because theoretically all possible combinations of signal pairs have to be checked. Therefore, BDD-based approaches such as functional learning [16] restrict their search to potential learning areas, which are identified by non BDD-based implication. Consequently, structural or hybrid approaches, i.e. BDDs combined with other methods, are predominant in ATPG, equivalence checking, and optimization of netlists. Even though most of these approaches make heavy use of implications, the data structures that are used for deriving and evaluating implications are often suboptimal and inflexible. That is why we propose a flexible data structure which is specifically optimized with respect to implication.

In this paper, we introduce a framework for implication based algorithms which inherits the advantages of structural as well as SAT-based approaches. Our approach combines both the flexibility and elegance of a SAT-based algorithm and the efficiency of a structural method by working on a graph model of the clause system, called *implication graph*. Its memory complexity is only linear in the number of modules in the circuit. Due to structural information available in the graph, fundamental problems such as justification, propagation, and particularly implication are carried out efficiently on the graph. The search for indirect implications reduces to graph algorithms that can be executed very fast and are easily extended to exploit bit-parallelism. As the implication graph can automatically be generated for any arbitrary logic, all presented algorithms remain valid independent of the chosen logic. This allows rapid prototyping of implication based tools for new multi-valued logics.

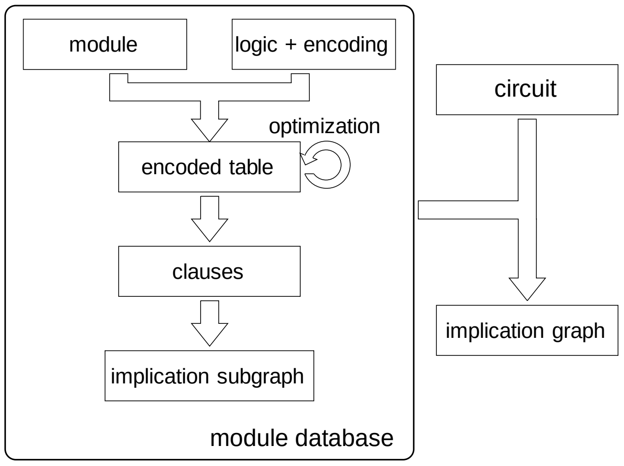
The remainder of this paper is organized as follows. In Section 2, we show how to derive the implication graph. Next, we discuss how implications are evaluated and how indirect implications can be computed in Sections 3 and 4, respectively. In order to demonstrate the high efficiency of our approach, experimental results for various applications using the proposed implication engine are presented in Section 5. Section 6 concludes the paper.

2 Implication graph

As performing implications is one of the most prominent and time consuming tasks in ATPG, equivalence checking, and optimization of netlists, it is of utmost importance to use a data structure that is best suited. Unlike other graphical representations of clause systems, our data structure represents the complete information contained in both the structural netlist and the clause database. The implication graphs used in NEMESIS[17] and TRAN[19] model only binary clauses, clauses of a higher order are solely included in the clause database.

Since our approach is generic in nature, any combinational circuit can automatically be compiled into its implication graph representation. Only information about a logic and its encoding as well as the truth table descriptions of supported module types have to be provided. The basic steps of compilation are given in Figure 1. First, all supported module types are individually compiled into encoded truth tables. Then, these tables are optimized by a two-level logic optimizer, e.g. ESPRESSO. This step is explained in Section 2.1. Next, a set of clauses is extracted from the optimized table, which is illustrated in Section 2.2. As shown in Section 2.3, the set of clauses is transformed into an implication subgraph that is stored in the module database. Then, for every module in the circuit the appropriate generic subgraph is taken from the module database and personalized with the input and

Figure 1 Deriving the implication graph



output signals of the given module. Finally, all identical nodes are merged into a single node resulting in the complete implication graph.

The following sections only consider the 3-valued logic $\mathcal{L}_3 = \{0, 1, X\}$ in order to present the basic ideas of our approach. Generation of an implication graph for an arbitrary multi-valued logic, e.g. the 10-valued logic $\mathcal{L}_{10}$ known from robust path delay ATPG, is discussed in Appendix A.

2.1 Encoding

A signal variable $x \in \mathcal{L}_3$ requires two encoding bits c_x and c_x^* for its internal representation. The complete scheme of encoding for $\mathcal{L}_3$ is shown in Table 1. In order to easily detect inconsistencies,

$x \in \mathcal{L}_3$	encoding		interpretation
	c_x	c_x^*	
0	0	1	signal x is 0
1	1	0	signal x is 1
X	0	0	signal x is unknown
	1	1	conflict at signal x

Table 1: 3-valued logic and its encoding

conflicting signal assignments are denoted by $c_x = 1 \wedge c_x^* = 1$. This property is expressed in the following definition:

DEFINITION 1 *An assignment is called non-conflicting iff $c_x \wedge c_x^* \Leftrightarrow 0$ holds for all signal variables x.*

Based on this encoding, the truth tables of all supported module types are converted into encoded tables. For example, the truth table of a 2-input AND-gate ($c = AND(a, b)$) found in Table 2 is converted into the encoded table of Table 2. This encoded table can be interpreted as specifying the

truth table		
a	b	c
0	0	0
0	1	0
1	0	0
1	1	1
0	X	0
1	X	X
X	0	0
X	1	X
X	X	X

encoded table					
c_a	c_a^*	c_b	c_b^*	c_c	c_c^*
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	0	1
1	0	1	0	1	0
0	1	0	0	0	1
1	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	0	0	0	0
1	1	-	-	-	-
-	-	1	1	-	-

optimized table					
c_a	c_a^*	c_b	c_b^*	c_c	c_c^*
-	1	-	-	-	1
-	-	-	1	-	1
1	-	1	-	1	-

Table 2: AND-gate: truth table — encoded table — optimized table

on-set as well as the off-set of two Boolean functions c_c and c_c^*.[1] Conflicting assignments belong to the don't-care-set, as they are explicitly checked for by the implication engine. Exploiting these don't-cares, functions c_c and c_c^* in the encoded table are optimized by ESPRESSO. The resulting optimized implicant table is given in Table 2.

2.2 Clause description

The characteristic function describing the AND-gate with respect to the given encoding can easily be derived in its *Conjunctive Normal Form (CNF)* by analyzing the individual rows of the optimized table of Table 2. Every row in the optimized implicant table corresponds to a clause contained in the *CNF*. Here, the *CNF* comprises the three clauses $\neg c_a^* \vee c_c^*$, $\neg c_b^* \vee c_c^*$, and $\neg c_a \vee \neg c_b \vee c_c$. That is, all valid value assignments to the inputs and outputs of the AND-gate are implicitly given by the satisfying assignments to the characteristic equation $CNF \Leftrightarrow 1$:

$$(\neg c_a^* \vee c_c^*) \wedge (\neg c_b^* \vee c_c^*) \wedge (\neg c_a \vee \neg c_b \vee c_c) \Leftrightarrow 1 \tag{1}$$

Of course, these satisfying assignments have to obey the constraint of being non-conflicting.

In Appendix A, it is shown how such a *CNF* is automatically derived for an arbitrary logic. In case of the 3-valued logic $\mathcal{L}_3$, the characteristic equation can be derived much simpler by computing the *CNF* of formula $a \wedge b \leftrightarrow c$ with respect to $\mathcal{L}_2 = \{0,1\}$:

$$c \leftrightarrow a \wedge b \Leftrightarrow (a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \tag{2}$$

If we replace a by $\neg c_a^*$ (b by $\neg c_b^*$), $\neg a$ by $\neg c_a$ ($\neg b$ by $\neg c_b$) and c by c_c ($\neg c$ by c_c^*) in Equation (2) we obtain Equation (1). Please note that this straightforward approach only works for $\mathcal{L}_3$. Our generic procedure, however, is valid for any arbitrary multi-valued logic as shown in Appendix A.

[1]The encoded table can also be seen as representing a list of multiple-output implicants for multi-output function $f = [c_c, c_c^*]$.

2.3 Building the implication graph

By exploiting the following equivalencies, the clause description of Equation (1) is converted into the corresponding implication graph[2].

$$x \vee y \quad \Leftrightarrow \quad (\neg x \rightarrow y) \wedge (\neg y \rightarrow x) \tag{3}$$

$$x \vee y \vee z \quad \Leftrightarrow \quad (\neg x \wedge \neg y \rightarrow z) \wedge (\neg x \wedge \neg z \rightarrow y) \wedge (\neg y \wedge \neg z \rightarrow x) \tag{4}$$

It is sufficient to provide equivalencies for binary and ternary clauses only, as any clause system of a higher order can be decomposed into a system of binary and ternary clauses by introducing additional variables[3] according to the following equivalence:

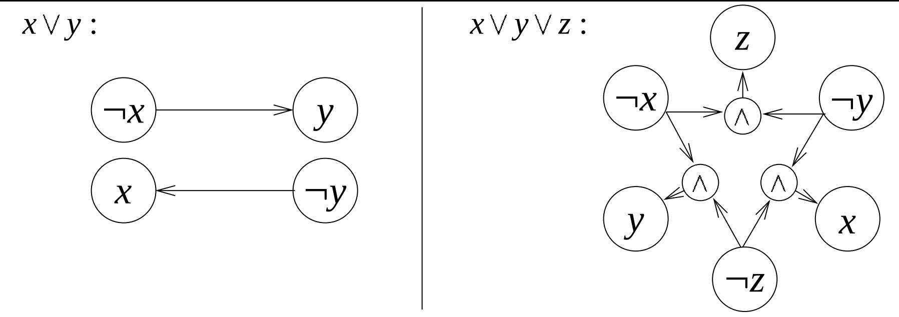
$$(w \vee x \vee y \vee z) \Leftrightarrow \underset{\nu \in \{0,1\}}{\exists} (w \vee x \vee \nu) \wedge (y \vee z \vee \neg \nu) \tag{5}$$

Introducing existential quantification is not problematic, as we are asking for satisfiability of a Boolean formula $f(\underline{x})$, i.e. $\underset{\underline{x}}{\exists} f(\underline{x})$. The existential quantifier in Equation (5) can be ignored when building the implication graph as the following equivalence holds:

$$\underset{\underline{x}}{\exists}[f(\underline{x}) \wedge \underset{z}{\exists} g(\underline{x}, z)] \Leftrightarrow \underset{\underline{x}, z}{\exists} f(\underline{x}) \wedge g(\underline{x}, z) \tag{6}$$

Having transformed all clauses into binary and ternary clauses, the subgraphs shown in Figure 2 are used for representation of these clauses. These graphs contain two types of nodes. While the first type

Figure 2 Implication subgraph for a binary and a ternary clause



represents the encoded signal values, the second one symbolizes the conjunction operation. The latter type is depicted by $\wedge$ or a shaded triangle[4]. Every ternary clause has three associated $\wedge$-nodes that uniquely represent the ternary clause in the implication graph.

Coming back to the 2-input AND-gate, its *CNF*-description is transformed into the implication graph shown in Figure 3. Every bit of the encoding for a signal x is represented by a corresponding

[2]Symbol "$\rightarrow$" denotes the conditional connective or implication in propositional logic.

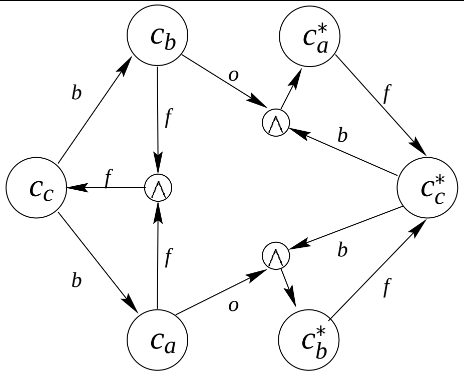
[3]This is very similar to decomposing a circuit having multi-input modules into one with 2-input modules only.

[4]We use two different symbols for these nodes as some of the graphs are automatically generated by our tool using the program DOT [21] from AT&T Bell Labs, Murray Hill,NJ.

node in the implication graph, e.g. node $c_a(c_a^*)$ in Figure 3 gives bit $c_a(c_a^*)$ of signal a. As we require non-conflicting assignments, literals $\neg c_x$ ($\neg c_x^*$) can be replaced by c_x^* (c_x) such that only nodes corresponding to non-negated encoding bits are contained in Figure 3.

Formally, the implication graph can be given by $G_I = (V_I, E_I)$ with V_I giving the nodes and E_I denoting the edges in the graph. V_I can be further partitioned into the set V_{IS}, which contains the nodes representing the encoded signal bits, and set $V_{I\wedge}$, which comprises the $\wedge$-nodes necessary to model ternary clauses[5]. So far, the implication graph only captures the logic functionality of a circuit. Since structural information is indispensable for some tasks, such as justification and propagation, we provide this information within the implication graph by marking its edges with three different tags f, b, and o. Edges that denote an implication from an input to an output signal of a module are marked with f (forward edge). Relations from output to input signals are tagged with b (backward edge). All other edges, e.g. input to input relations and indirect implications, are given tag o (other edge)[6]. The tags for the 2-input AND-gate are found in Figure 3. By means of these tags, a directed acyclic graph

Figure 3 Implication graph for 2-input AND-gate



(DAG) can be extracted from the implication graph. If all edges but the forward edges are removed, we obtain a DAG that forms the base of an efficient algorithm for backtracing and justification.

For a simple circuit, the different circuit descriptions introduced above are compiled in Example 2.1. Please observe that most clause based approaches work on a *CNF* in $\mathcal{L}_2$ while our framework operates on a *CNF* of variables encoded with respect to a given logic, here $\mathcal{L}_3$. A structural description provides information about the logic behavior of the circuit by specifying the module types, which implement some logic function, and their interconnection. A clause description (*CNF*), on the contrary, captures the complete information in a single Boolean logic formula. In order to make clear, which modules contribute which clauses to the *CNF*, we have indicated the corresponding modules on the right hand side of the respective *CNF*. The implication graph of the example circuit is given in Figure 4. It captures information on the logic behavior as well as the structure of the circuit in one

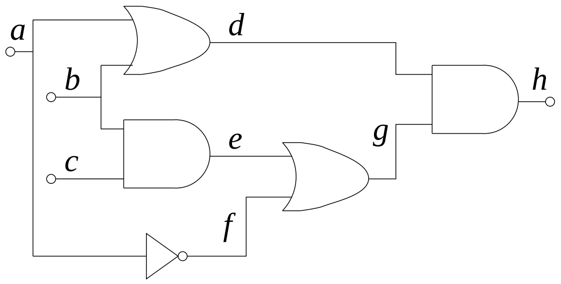
[5]The implication graphs proposed in [17, 19] form a subgraph of our implication graph, as they model only binary clauses.

[6]Tags denoting other edges have been omitted in later examples.

graph. The dashed and dotted edges in Figure 4 represent indirect implications that can be deduced by the techniques presented in Section 4.

Example 2.1 Circuit descriptions: structural — clauses — implication graph

- Structural:



- *CNF* for $\mathcal{L}_2$:

$$\begin{array}{ll|l} CNF_2 & \Leftrightarrow & \\ (d \vee \neg h) \wedge (g \vee \neg h) \wedge (\neg d \vee \neg g \vee h) & \wedge & h = AND(d,g) \\ (\neg a \vee d) \wedge (\neg b \vee d) \wedge (a \vee b \vee \neg d) & \wedge & d = OR(a,b) \\ (\neg e \vee g) \wedge (\neg f \vee g) \wedge (e \vee f \vee \neg g) & \wedge & g = OR(e,f) \\ (b \vee \neg e) \wedge (c \vee \neg e) \wedge (\neg b \vee \neg c \vee e) & \wedge & e = AND(b,c) \\ (a \vee f) \wedge (\neg f \vee \neg a) & & f = NOT(a) \\ & \Leftrightarrow 1 & \end{array}$$

- *CNF* for $\mathcal{L}_3$:

$$\begin{array}{ll|l} CNF_3 & \Leftrightarrow & \\ (\neg c_d^* \vee c_h^*) \wedge (\neg c_g^* \vee c_h^*) \wedge (\neg c_d \vee \neg c_g \vee c_h) & \wedge & h = AND(d,g) \\ (\neg c_a \vee c_d) \wedge (\neg c_b \vee c_d) \wedge (\neg c_a^* \vee \neg c_b^* \vee c_d^*) & \wedge & d = OR(a,b) \\ (\neg c_e \vee c_g) \wedge (\neg c_f \vee c_g) \wedge (\neg c_e^* \vee \neg c_f^* \vee c_g^*) & \wedge & g = OR(e,f) \\ (\neg c_b^* \vee c_e^*) \wedge (\neg c_c^* \vee c_e^*) \wedge (\neg c_b \vee \neg c_c \vee c_e) & \wedge & e = AND(b,c) \\ (\neg c_a^* \vee c_f) \wedge (\neg c_f \vee c_a^*) & & f = NOT(a) \\ & \Leftrightarrow 1 & \end{array}$$

- Implication graph for $\mathcal{L}_3$: (See Figure 4)

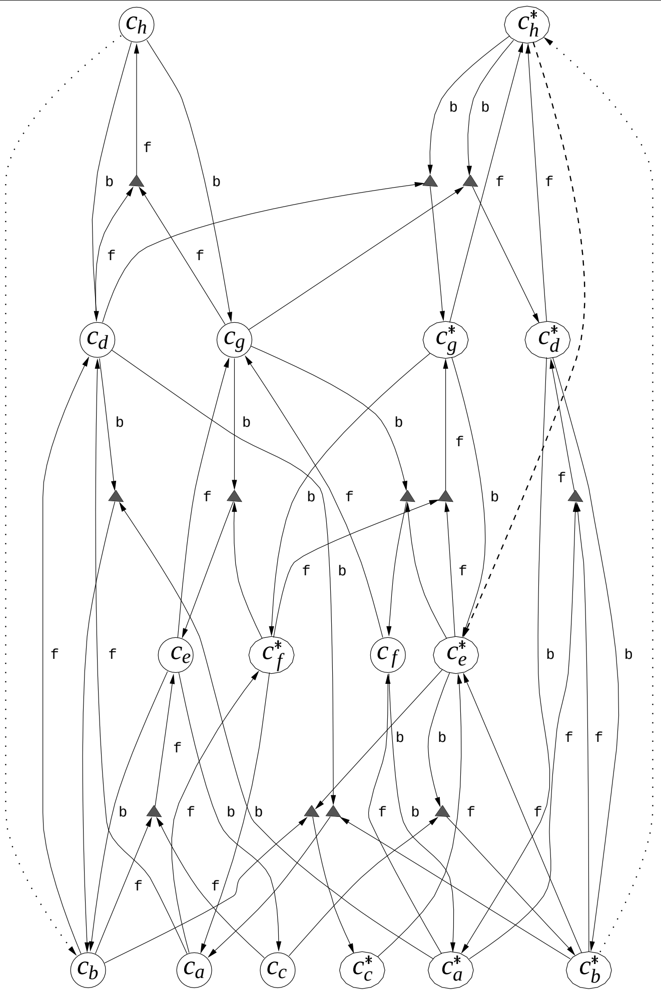
2.4 Advantages

Using the proposed implication graph as a core data structure in CAD algorithms has many advantages:

1) Important tasks such as implication and justification can be carried out on the implication graph in the same manner for any arbitrary logic. The peculiarities of the chosen logic are included in the graph. Implication and derivation of indirect implications reduce to efficient graph algorithms as will be shown in Sections 3.3 and 4.4.

2) Most SAT-based algorithms use a static order for variable assignments during their search for a satisfying assignment [17, 19]. Furthermore, these algorithms assign values to internal signals during justification. Since PODEM, it has been well known that assigning values only to primary input signals helps to reduce the search space. Obviously, primary inputs are a special property of the given instance of SAT which is not exploited by algorithms for solving arbitrary SAT

Figure 4 Implication graph of circuit in Example 2.1



problems. The algorithm of TEGUS tries to mimic PODEM by ordering the clauses in a special manner[18]. Our approach does not need such additional techniques, as structural information is provided by the edge tags.

3) Algorithms working on the implication graph can easily exploit bit-parallelism as the status of every node can be represented by one bit only. For example, on a 64-bit machine 64 implication graphs can be processed in parallel, making bit-parallel implication very efficient.

4) Sequential circuits are often modelled as an iterative logic array (ILA). In this model the time domain is unfolded into multiple copies of the combinational logic block. These logic blocks can be compiled into the corresponding implication graphs. Using bit-parallel techniques, a 64-bit machine allows to keep 64 time-frames without increasing the size of the implication graph.

3 How to perform implications

3.1 Structure based

Structure based implication is a special form of event-driven simulation. Contrary to ordinary simulation, which starts at the primary inputs, implication is started at an arbitrary signal in the circuit. Therefore, it has to proceed towards the primary outputs as well as the primary inputs such that implications are often categorized into forward and backward implications. Obviously, this technique requires many table lookups for evaluating the module functions. This becomes particularly costly for multi-valued logics, e.g. the ones used in path delay ATPG.

Let us consider the circuit given in Example 2.1. If we assign logical 0 to signal g, $e = 0$, $f = 0$, $a = 1$, $d = 1$ follows by backward implication, and $h = 0$ is deduced by forward implication.

3.2 Clause based

Clause based implication relies on *Boolean Constraint Propagation (BCP)*. BCP corresponds to an iterative application of the unit clause rule proposed by Davis et al. in 1960 [22]. In BCP, unary clauses are used to simplify other clauses until no further simplification is possible or some clause becomes unsatisfied. Implication is started by adding a unary clause, which represents the initial signal assignment, to the CNF. All unary clauses computed by BCP correspond to implications from the initial assignment as they force the corresponding signals to a certain logic value.

Let us explain BCP with the help of CNF_2 in Example 2.1. If we assign 0 to signal g, all clauses containing literal $\neg g$ become satisfied and clauses containing g are reduced to either binary or unary clauses. Next, the computed unary clauses, i.e. $\neg h$, $\neg e$, $\neg f$, are used to generate additional unary clauses. After termination of BCP, the following unary clauses are deduced: $\neg h$, $\neg e$, $\neg f$, a, d. As these unary clauses force the corresponding signals to a certain logic value, they represent implications that follow from the initial assignment $g = 0$.

The most time consuming task in BCP is the search for clauses that can be simplified by the unit clause rule. This search is not necessary when working on the implication graph since clauses that share common variables are connected in the graph.

3.3 Implication graph based

Implication graph based implication is simple and efficient, as it only requires a partial traversal of the implication graph. In TRAN [19], this traversal is done implicitly by computing the transitive closure of the implication graph. As this implication graph only represents binary clauses it has to be updated if a ternary clause reduces to a binary clause which results in considerable computational costs.

Implying from a signal assignment means that first the corresponding nodes[7] are marked in the implication graph. Then the implication procedure traverses the implication graph obeying the following

[7]An assignment to a single signal causes the corresponding encoding bits for the chosen multi-valued logic to be set. Thus, a singular signal assignment may result in several marked nodes in the implication graph.

rule:

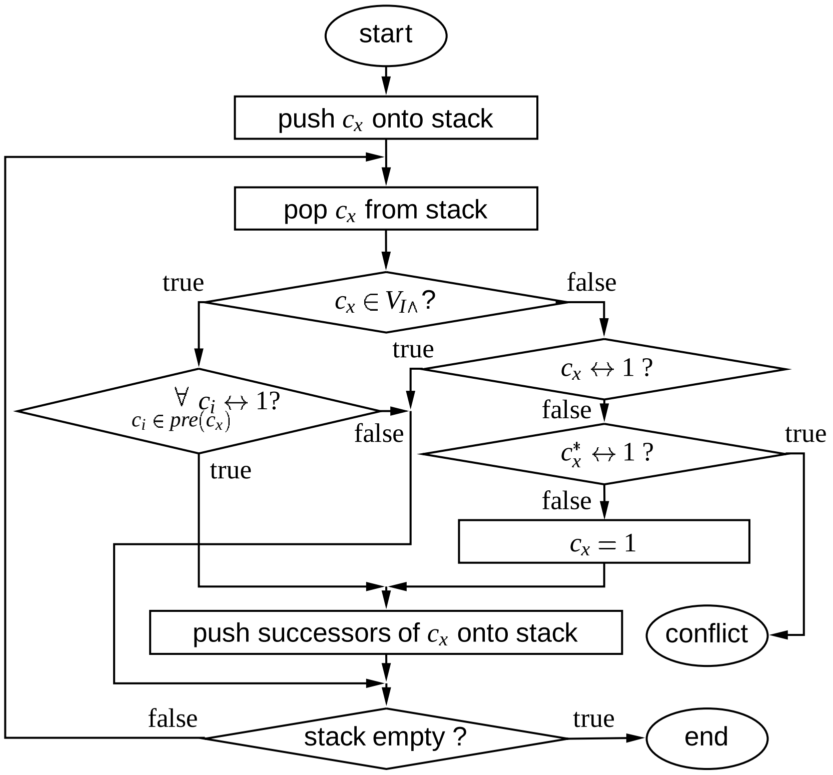
RULE 1 *Starting from an initial set S_I of marked nodes, all successor nodes s_j are marked*

- *if node s_j is a $\wedge$-node and* ***all*** *its predecessors are marked.*
- *if node s_j represents an encoding bit and* ***at least one*** *predecessor is marked .*

This rule is applied until no further propagation of marks is possible.

Algorithm 1 gives a flow chart of a procedure `imply(`c_x`)` which implements Rule 1 using a stack. All

Algorithm 1 imply(c_x)



nodes that have been marked represent signal values that can be implied from the initial assignment given by S_I. Conflicting signal assignments are easily detected during implication, since they cause both nodes c_x and c_x^* to be marked.

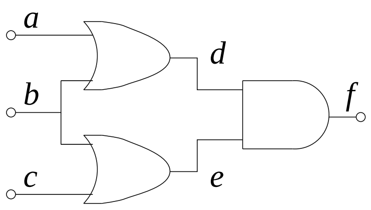
Again we use the circuit of Example 2.1 for the sake of explanation. Assigning logical value 0 to g corresponds to marking node c_g^* in the implication graph of Figure 4. After running the implication procedure, the following nodes are marked: c_h^*, c_e^*, c_f^*, c_a, c_d. So as to obtain the implied signal value with respect to the given logic, the marked nodes, which correspond to set encoding bits, are decoded according to the given encoding, i.e. we determine $h = 0$, $e = 0$, $f = 0$, $a = 1$ and $d = 1$. It can be seen that the set of marked nodes given above corresponds to the signal values computed by the other approaches.

4 Deriving indirect implications

Contrary to direct implications, detection of indirect implications requires a special analysis of the logic function of a circuit as they represent information on the circuit that is not obvious from its description. Most methods for computation of indirect implications are subject to order dependency. That is, some indirect implications can only be found if certain other indirect implications have already been discovered. In order to avoid this problem, it has been suggested to iterate their computation [18].

Example 4.1 Circuit descriptions: structural — clauses — implication graph

- Structural:



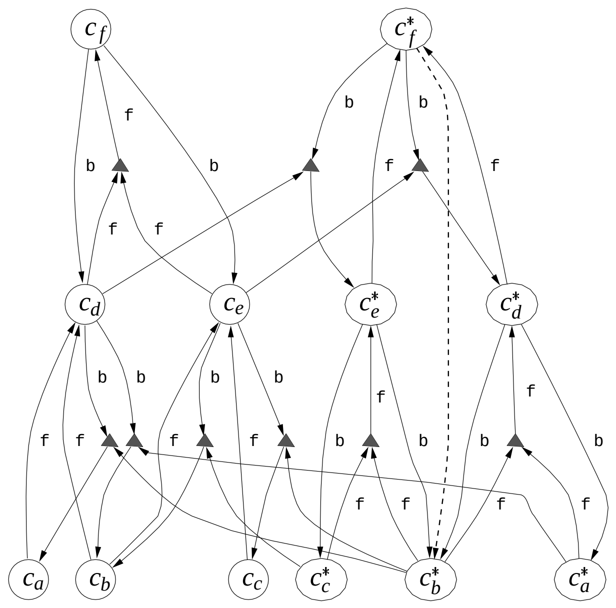
- *CNF* for $\mathcal{L}_2$:

$$\begin{array}{lcl} CNF_2 & \Leftrightarrow & \\ (d \vee \neg f) \wedge (e \vee \neg f) \wedge (\neg d \vee \neg e \vee f) \quad \wedge & & f = AND(d,e) \\ (\neg a \vee d) \wedge (\neg b \vee d) \wedge (a \vee b \vee \neg d) \quad \wedge & & d = OR(a,b) \\ (\neg b \vee e) \wedge (\neg c \vee e) \wedge (b \vee c \vee \neg e) \quad \wedge & & e = OR(b,c) \\ & \Leftrightarrow 1 & \end{array}$$

- *CNF* for $\mathcal{L}_3$:

$$\begin{array}{lcl} CNF_3 & \Leftrightarrow & \\ (\neg c_d^* \vee c_f^*) \wedge (\neg c_e^* \vee c_f^*) \wedge (\neg c_d \vee \neg c_e \vee c_f) \quad \wedge & & f = AND(d,e) \\ (\neg c_a \vee c_d) \wedge (\neg c_b \vee c_d) \wedge (\neg c_a^* \vee \neg c_b^* \vee c_d^*) \quad \wedge & & d = OR(a,b) \\ (\neg c_b \vee c_e) \wedge (\neg c_c \vee c_e) \wedge (\neg c_b^* \vee \neg c_c^* \vee c_e^*) \quad \wedge & & e = OR(b,c) \\ & \Leftrightarrow 1 & \end{array}$$

- Implication graph for $\mathcal{L}_3$:



4.1 Structure based

The SOCRATES algorithm [13] was the first to introduce computation of indirect implications using the following tautologies:

$$(a \rightarrow b) \quad \Leftrightarrow \quad (\neg b \rightarrow \neg a) \tag{7}$$

$$(a \rightarrow b) \wedge (a \rightarrow \neg b) \quad \Rightarrow \quad \neg a \tag{8}$$

While Equation (7) (*law of contraposition* or *contrapositive*) may generate a candidate for an indirect implication, Equation (8) identifies a fix value.

Indirect implications are primarily computed in a pre-processing phase. The idea is to temporarily set a given signal to a certain logic value. Then, all possible direct implications from this signal assignment are computed. For all implied signal values, it is checked if the contrapositive cannot be deduced by direct implications (*learning criterion*). In this case, the contrapositive is an indirect implication. As indirect implications cannot be represented within the data structure used to describe the circuit, structural algorithms have to store them in an external data structure. This adds additional complexity to structure based algorithms.

For the circuit given in Example 4.1, computation of indirect implications with respect to logic $\mathcal{L}_3$ may be started by injecting a logical 1 at fanout signal b. Executing all possible direct implications yields $d = 1$, $e = 1$, and $f = 1$. The contrapositive of $b \rightarrow f$ gives the indirect implication $\neg f \rightarrow \neg b$, which is stored.

4.2 Clause based

Clause based computation [17, 18] is similar to the structural algorithm of Section 4.1. Each free literal a contained in the *CNF* is temporarily set to 1. Then BCP is used to derive all possible direct implications, i.e. unary clauses. For all generated unary clauses b, it is checked if the contrapositive $\neg b \rightarrow \neg a$ is an indirect implication. In this case, the corresponding clause $b \vee \neg a$ is added to the clause database. Thereby, indirect implications enrich the data structure used for representing the circuit functionality. Once an indirect implication has been added to the clause database, it does no longer require any special attention. This is one important advantage of clause based algorithms over structure based approaches [18].

Let us explain clause based computation of indirect implications with CNF_2 of Example 4.1. Setting b to 1 and running BCP yields unary clauses d, e, and f. Only the contrapositive of $b \rightarrow f$, i.e. $\neg f \rightarrow \neg b$, forms an indirect implication such that clause $f \vee \neg b$ is added to the *CNF*.

4.3 AND-OR enumeration

A different approach, known as recursive learning, has been taken by Kunz et al. [3, 7]. Indirect implications are deduced by an AND-OR search [23] for all possible implications resulting from a signal assignment. This search is performed by injecting and reversing signal assignments followed by deriving all direct implications. Only a simple structural algorithm for executing implications is applied. Unlike the methods presented so far, recursive learning is complete. Due to the exponential nature of this search, a complete search is, however, not feasible.

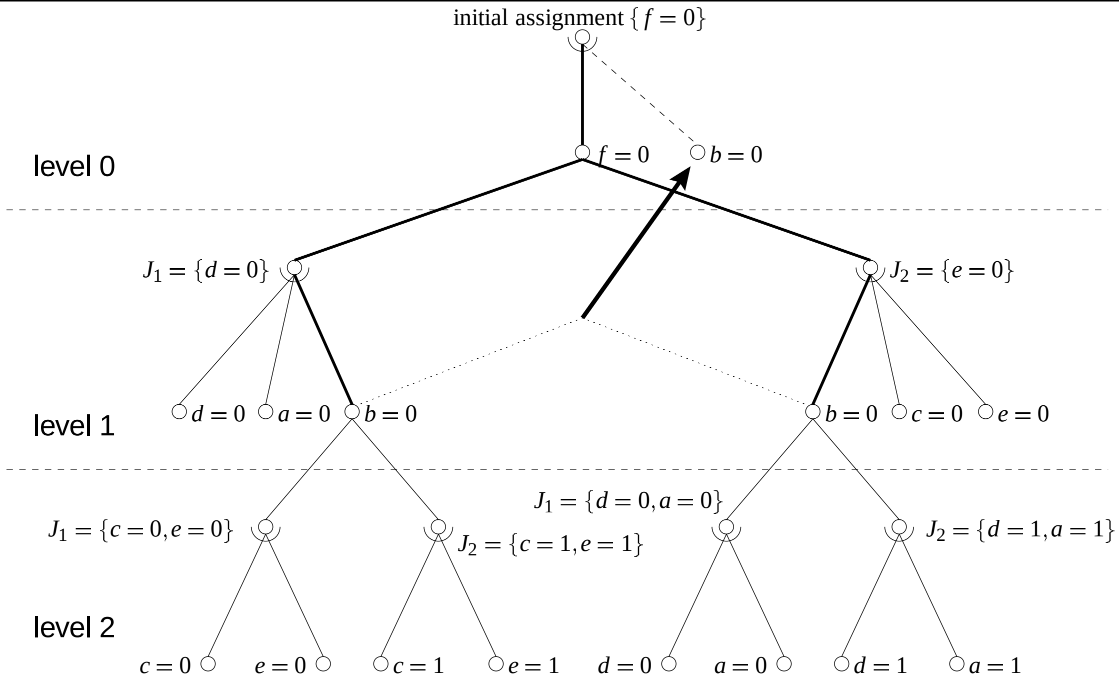
Recursive learning is based on the notions of unjustified gates and justifications. The following definitions are taken from [7].

DEFINITION 2 *Given a gate G in a combinational network that has at least one specified input or output signal and the values at G are logically consistent: Gate G is called unjustified, if there are one or several unspecified input or output signals of G for which there exists a combination of value assignments yielding a conflict at G. Otherwise G is called justified.*

DEFINITION 3 *Let $f_1, f_2, \ldots, f_n$ be some unspecified input or output signals of an unjustified gate G and let $V_1, V_2, \ldots, V_n$ be logic values that specify them. The set of signal assignments, $J = \{f_1 = V_1, f_2 = V_2, \ldots, f_n = V_n\}$, is called justification for G, if the combination of value assignments in J makes G justified.*

Let us illustrate the principles of the AND-OR enumeration with the circuit of Example 4.1 and the AND-OR tree found in Figure 5. The root node of the AND-OR tree reflects the initial assignment,

Figure 5 AND-OR enumeration



it is of the AND-type[8]. In our example, a logical 0 is assigned to signal f. As no further signal values can be implied, OR-node $f = 0$ is the only successor of the root node. The justifications for $f = 0$ are $J_1 = \{d = 0\}$ and $J_2 = \{e = 0\}$. In order to derive an indirect implication, we have to search for implied

[8]In general, an AND-node (marked by an arc) represents a signal assignment due to justification of an unjustified gate, whereas an OR-node denotes a signal value that can be implied from a chosen justification. Justified gates correspond to OR-leaves and unjustified gates to internal OR-nodes in the AND-OR graph [3].

signal values that are common to both justifications. That is, we have to solve both subproblems for solving the main problem, which is a typical feature of AND-OR search. Here, $b = 0$ is implied for both justifications. This is represented by a new OR-node $b = 0$ in level 0 of the AND-OR tree. In general, new OR-nodes in level 0 correspond to indirect implications. Further examination of gates in level 2, which have become unjustified because of setting b to 0, does not yield additional indirect implications.

In Example 2.1, the indirect implications $h \rightarrow b$ ($c_h \rightarrow c_b$) and $\neg b \rightarrow \neg h$ ($c_b^* \rightarrow c_h^*$) can only be determined by AND-OR search (indicated by dotted arrows in Figure 4). Indirect Implication $\neg h \rightarrow \neg e$ ($c_h^* \rightarrow c_e^*$) is also found by contraposition from direct implication $e \rightarrow h$ ($c_e \rightarrow c_h$)(indicated by a dashed arrow in Figure 4).

4.4 Implication graph based

An implication graph based method for computing indirect implications inherits all advantages of clause based techniques but eliminates the costly search process required during BCP-based implication.

The techniques presented in NEMESIS [17] and TRAN [19], however, remain limited in their capabilities as their implication graphs only represent binary clauses. Moreover, these approaches are not complete. In NEMESIS, a search for strongly connected components in the implication graph is used to identify equivalent (antivalent) signals or fix values. Additionally, NEMESIS also applies a clause based technique. TRAN derives indirect implications by computing the transitive closure of its implication graph and checking for certain properties in it. The restriction to binary clauses makes it necessary to dynamically update the graph once ternary clauses are reduced. This makes the proposed procedure complicated and costly.

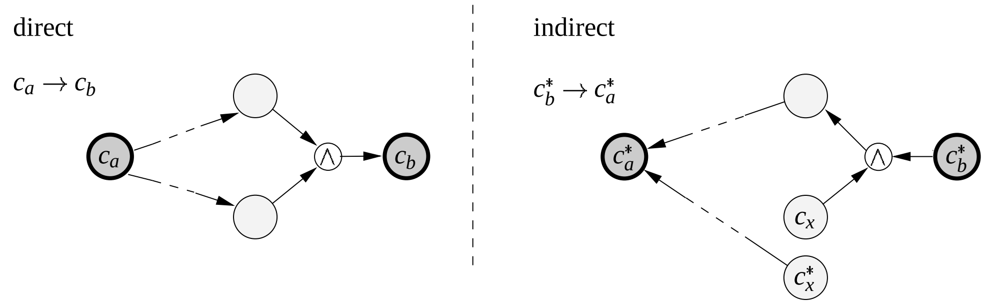
The implication graph based approach presented here does not suffer from this deficiencies. For the first time, computation of indirect implications based on the law of contraposition and AND-OR enumeration are integrated into the same framework by our approach.

4.4.1 Deducing indirect implications by reconvergence analysis

The basic idea of determining indirect implications by a search for reconvergencies is shown in Figure 6. While implication $c_a \rightarrow c_b$ is deduced by direct implication, $c_b^* \rightarrow c_a^*$ forms an indirect implication. This is due to the fact, that we have to traverse a reconvergent structure in the implication graph when implying $c_a \rightarrow c_b$. The $\wedge$-node can only be passed if both of its predecessors are marked, i.e. it forms a reconvergence $\wedge$-node during implication. If we start implication at node c_b^*, however, we cannot pass the $\wedge$-node, as its other predecessor c_x is not marked. Applying the law of contraposition to $c_a \rightarrow c_b$, we easily deduce $c_b^* \rightarrow c_a^*$ such that c_a^* is implied from c_b^*. This observation is expressed in the following lemma:

LEMMA 1 *Let c_x represent the initial assignment. A reconvergent structure (c_x, c_y) in the implication graph yields an indirect implication $c_y^* \rightarrow c_x^*$ only if*

Figure 6 Learning by contraposition on the implication graph



- *c_x is a fanout node in the implication graph.*
- *a node c_y is marked via a $\wedge$-node and both predecessors of the $\wedge$-node have been marked by implying along disjoint paths in the implication graph.*

Proof:

i) If the predecessors of the $\wedge$-node are not marked by implying along disjoint paths then there exists a fanout node c_f in the implication graph that can be implied from c_x by direct implication. Then, an assignment at node c_f causes both predecessors of the $\wedge$-node to be marked along disjoint paths and $c_y^* \rightarrow c_f^*$ is an indirect implication. $c_y^* \rightarrow c_x^*$ can then be deduced by direct implication $c_y^* \rightarrow c_f^*$ and $c_f^* \rightarrow c_x^*$.

ii) If c_x is no fanout node, then there exist no disjoint paths to a $\wedge$-node. □

Remark: Depending on the order in computing indirect implications, an originally indirect implication may become a transitive implication of a newly computed indirect implication and other implications. It could therefore be removed from the graph. Requiring disjoint paths helps not to consider such transitive implications. Of course, this technique is order dependent.

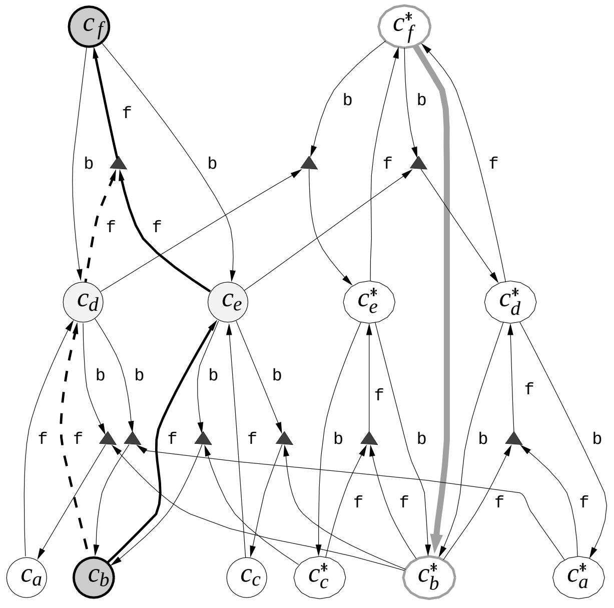
Reconvergent structures of the type defined in Lemma 1 are identified by an adapted version of the algorithm for reconvergence analysis presented in [24]. Using Lemma 1 it can be shown that the search for reconvergencies in the implication graph detects all indirect implications, which are found by clause and structural based approaches.

THEOREM 1 *All indirect implications found by BCP on the (encoded) clause description can be identified by a search for the reconvergent structures defined in Lemma 1.*

Proof: BCP-based implication requires the reduction of ternary or binary clauses contained in the *CNF* to unary clauses. Unary clauses gained by reducing binary clauses of the *CNF* do not yield candidates for indirect implications as the contrapositive can already be computed by BCP on the original *CNF*. In the implication graph, ternary clauses are represented by the structure given in Figure 2. The three possibilities for reducing this ternary clause to a unary clause are given by the three successor nodes of the three $\wedge$-nodes. These successor nodes and the corresponding $\wedge$-nodes form the reconvergent part according to Lemma 1. The ternary clause can only be reduced to a unary clause by BCP if the initial assignment generates at least two unary clauses. This corresponds to a marked fanout node in the implication graph. □

We explain the reconvergence analysis with Figure 7, which displays the implication graph of Example 4.1. Let's assume that fanout node c_b is marked. Then, the implication procedure of Section 3.3 is invoked. As both c_d and c_e have been marked, the succeeding $\wedge$-node and c_f are marked, too. The $\wedge$-node has been reached via two disjoint paths in the graph (indicated by the dashed and solid line, respectively) such that the contrapositive $c_f^* \rightarrow c_b^*$ forms an indirect implication. This indirect implication is included into the graph in form of the grey edge leading from node c_f^* to node c_b^*.

Figure 7 Reconvergence analysis on the implication graph



Applying our graph analysis offers the following advantages:

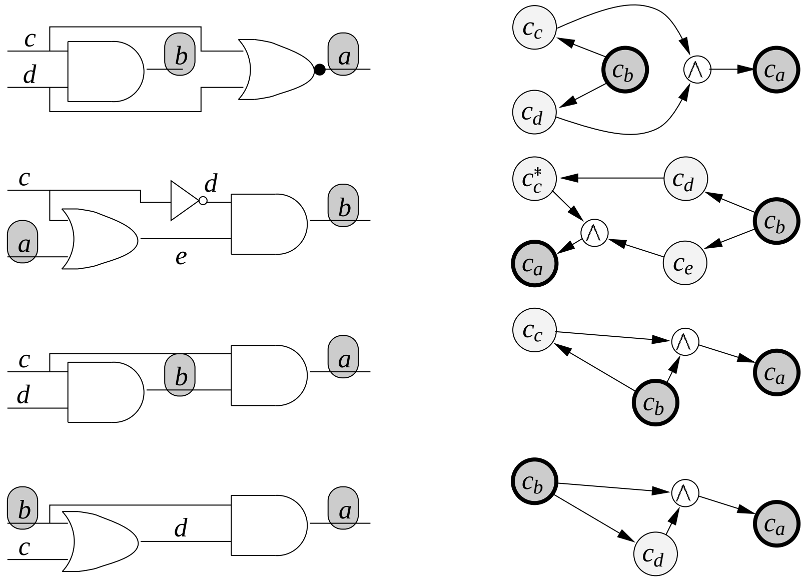
1) The search for reconvergence regions in the implication graph reduces the set of candidate signals that may yield an indirect implication. Clause based methods have to temporarily assign a value to all literals contained in the *CNF*.

2) Our method does not require a learning criterion such as the approach of [13].

3) Reconvergence analysis is carried out very fast by an adapted version of the algorithm presented in [24].

4) The reconvergence criterion of Lemma 1 could be adapted to a structural description, too. As can be seen in Figure 8, reconvergencies in the implication graph translate to various types of reconvergent structures in the structural description. Therefore, an efficient procedure that deals with all possible types of reconvergencies is difficult to devise for the structural description.

Figure 8 Types of structural reconvergencies and corresponding implication graphs



4.4.2 Deducing indirect implications by extended reconvergence analysis

Contrary to the reconvergence analysis of Section 4.4.1, the extended reconvergence analysis detects conditional reconvergencies at signal nodes. As it corresponds to an AND-OR search in the implication graph, we need the following definitions:

DEFINITION 4 *A clause $C = c_1 \vee c_2 \vee \ldots \vee c_n$ is called unjustified iff all literals $c_1, c_2, \ldots, c_n$ do not evaluate to 1 and at least one complement $\neg c_i$ of a literal c_i is 1.*

Unjustified ternary clauses are found in the implication graph without effort. They are represented by $\wedge$-nodes that have exactly one of their two predecessors marked.

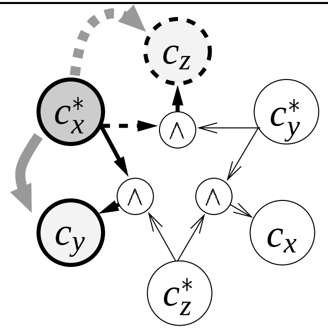
DEFINITION 5 *Let $c_1, c_2, \ldots, c_m$ be some unspecified literals in a clause $C = c_1 \vee c_2 \vee \ldots \vee c_n$ that is unjustified, and let $V_1, V_2, \ldots, V_m$ denote the assigned values. Then, the set of assignments $J = \{c_1 = V_1, c_2 = V_2, \ldots, c_m = V_m\}$ is called a justification*[9] *of clause C, if the value assignments in J make C evaluate to 1.*

[9]If the given clause consists of literals representing bits of a given encoding the justification has to be non-conflicting.

In a clause based framework a complete set of justifications J_c for an unjustified clause C is easily given by $J_c = \{\{c_1 = 1\}, \{c_2 = 1\}, \ldots, \{c_m = 1\}\}$. For our approach, set J_c is even simpler, as only ternary clauses can be unjustified[10]. Therefore, J_c always consists of exactly two justifications.

We will now explain how these two justifications can be derived in the implication graph with Figure 9. The given ternary clause $c_x \vee c_y \vee c_z$ is unjustified due to an assignment of $c_x^* = 1$. This is

Figure 9 Unjustified ternary clause $c_x \vee c_y \vee c_z$ due to assignment $c_x^* = 1$



indicated by the two $\wedge$-nodes that have exactly one predecessor (c_x^*) marked. Each successor of these two $\wedge$-nodes defines one justification in J_c. Here, the ternary clause can be justified by setting c_z or c_y to 1. If we consider that the subgraph denoting the ternary clause $c_x \vee c_y \vee c_z$ is a straightforward graphical representation of the following formulae

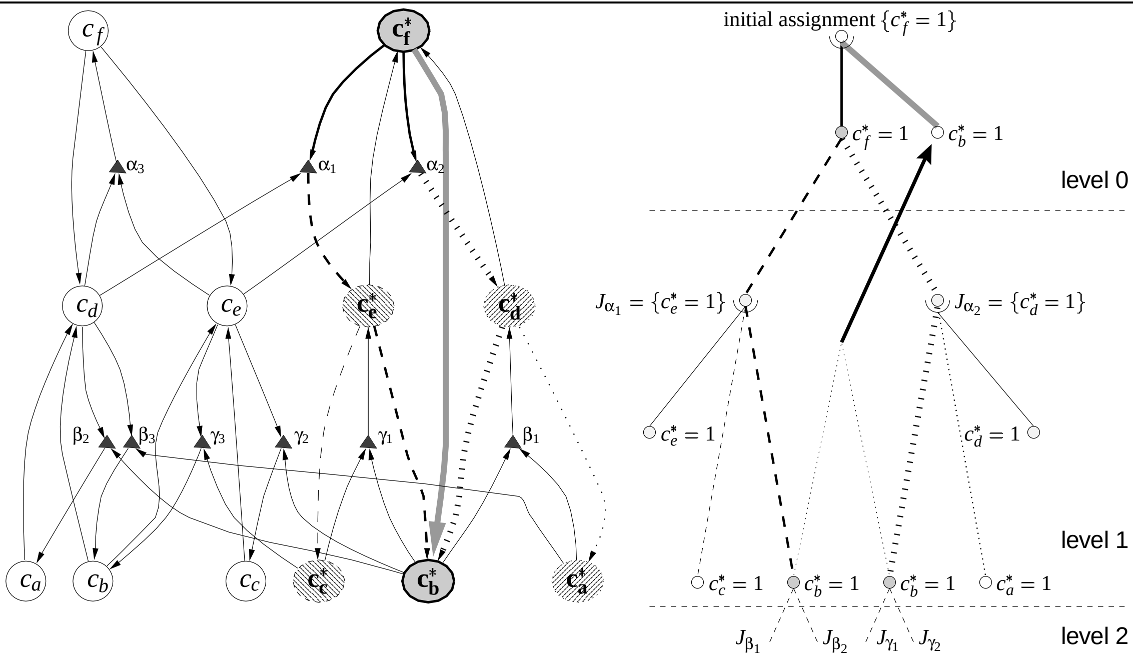
$$c_x^* \wedge c_y^* \to c_z \Leftrightarrow c_x^* \wedge c_z^* \to c_y \Leftrightarrow c_y^* \wedge c_z^* \to c_x$$

it becomes apparent that both possible justifications are found in the consequents of those implications which have the literal making the clause unjustified, i.e. c_x^*, in their antecedent. These consequents correspond to the successors of the $\wedge$-nodes.

Let us now explain how the extended reconvergence analysis corresponds to an efficient AND-OR search on the implication graph with help of Figure 10 showing the implication graph of Example 4.1. An initial assignment of $c_f^* = 1$ makes clause $C_\alpha = c_d^* \vee c_e^* \vee c_f$ unjustified. Next, the possible justifications $J_{\alpha_1} = \{c_e^* = 1\}$, $J_{\alpha_2} = \{c_d^* = 1\}$ for C_α are determined as the successors of the two $\wedge$-nodes α_1 and α_2 belonging to clause C_α. These $\wedge$-nodes correspond to AND-nodes J_{α_1} and J_{α_2} in the AND-OR tree, respectively. So as to distinguish between the consequences of the two justifications, each one is assigned a different color. Thus, node $c_e^* = 1$ is given a green marker (represented by dashed lines in Figure 10) and all signals that can be implied from $c_e^* = 1$ are marked green. The same is done for $c_d^* = 1$ using a red marker (dotted lines in Figure 10). Nodes that are assigned both colors, i.e. nodes where the markers reconverge, can be implied independent of the chosen justification. These nodes can therefore be elevated to the previous level in the AND-OR tree. In our example, only node c_b^* is marked by both colors and we derive the indirect implication $c_f^* \to c_b^*$. Further analysis of unjustified clauses C_β and C_γ in level 2 of the AND-OR tree does not yield additional indirect implications.

[10]If a binary clause is unjustified according to Definition 4, it reduces to a unary clause. Unary clauses represent necessary assignments (implied signal values) for the given signal assignment.

Figure 10 Extended reconvergence analysis on the implication graph



This example indicates that the trace of the extended reconvergence analysis is identical to the AND-OR tree generated by AND-OR enumeration if marked $\wedge$-nodes are converted to AND-nodes and marked signal nodes to OR-nodes. Obviously the extended reconvergence analysis is capable of determining all indirect implications given enough colors, i.e. it is complete.

An efficient procedure implementing this extended reconvergence analysis is given in Algorithm 2. It takes advantage of the implication graph by encoding the colors locally at the nodes using only bit slices of a full machine word. Thus, subtrees of the AND-OR tree are stored in parallel in different bit-levels. Additionally, `imply(node,mask)`, a bit-parallel version of the implication algorithm introduced in Section 3.3, is used. Our algorithm supports a depth of r levels in the AND-OR tree on a 2^r-bit architecture. On a DECAlphaStation, for example, a maximal depth of 6 levels is available.

Let us briefly summarize the advantages of our approach:

1) The implication graph model allows the full word size to be exploited by means of bit-parallel techniques. The search for indirect implications, requires efficient set operations as an OR-node may only be elevated if it is a successor of both AND-nodes belonging to an unjustified clause. These set operations are carried out very efficiently on the implication graph by performing local bit-operations at signal nodes such that no separate data structure is needed. Please note, that the advantage of highly efficient set operations remains, if we extend our algorithm to handle arbitrary depths of AND-OR enumeration, which has already been done.

2) The notion of unjustified gates necessary in [3, 7] reduces to the simple and elegant concept of unjustified ternary clauses. Due to this concept and the uniformity of our description, AND-OR enumeration can easily be performed for arbitrary logics applying the same procedure. This

Algorithm 2 AND-OR_enumerate(node, r, mask)

```
conflict = imply(node,mask)
if conflict then
    reset value assignments done by imply
    return conflict
else if r=0 then
    return 0
end if
r = r-1
for all unjustified clauses do
    find justifications $J_1$,$J_2$
    mask_1 = (mask $\ll 2^r$) && mask
    conflict = AND-OR_enumerate($J_1$,r,mask_1)
    if conflict then
        conflict = AND-OR_enumerate($J_2$,r,mask)
        if conflict then
            reset value assignments done during this invocation of AND-OR_enumerate
            return conflict
        end if
        continue with next unjustified clause
    end if
    mask_2 = (mask $\gg 2^r$) && mask
    conflict = AND-OR_enumerate($J_2$,r,mask_2)
    if conflict then
        update masks of node assignments done by AND-OR_enumerate($J_1$,r,mask_1) to mask
        continue with next unjustified clause
    end if
    reset nodes with ($val$(node) && mask) $\neq$ mask
    continue with next unjustified clause
end for
return 0
```

has already been done for logic $\mathcal{L}_{10}$. Higher valued logics are complicated to deal with in the structural approach of [7, 3].

3) Detected indirect implications can be included into the graph immediately. These indirect implications often facilitate the computation of other indirect implications.

4) Some indirect implications are easily computed by the law of contraposition while requiring a high depth of AND-OR search. As our approach integrates both methods into the same framework, indirect implications are automatically identified by the best suited technique.

5 Experimental results

The implication engine, presented in this paper, has been implemented in a C language library of functions that has been applied successfully to several CAD problems. Note that some of the presented results have already been published in papers dealing with application specific issues. The underlying implication engine was not discussed. We have included these results in order to show the efficiency of our flexible approach. While the experiments for ATPG and netlist optimization were carried out on a DECStation3000/600, the experiments for equivalence checking were performed on a DECAlphaStation250$^{4/266}$. ATPG and netlist optimization rely on an earlier version of our implication engine, that does not support the techniques of Section 4.4.2. So far, these advanced techniques have only been used for equivalence checking.

Tables 3 and 4 present results for ATPG considering various fault models [25, 26, 27, 28]. Due to the high flexibility of the implication graph the various logics ($\mathcal{L}_3, \mathcal{L}_9, \mathcal{L}_{10}, \mathcal{L}_{20}$) required for the different fault models could easily be handled. Table 3 gives the number of tested faults and CPU time for gate-delay faults in combinational circuits and non-robust as well as robust path delay faults in sequential circuits using only a standard scan design. Table 4 shows the corresponding results for both types of path delay faults and stuck-at faults in combinational circuits (or sequential circuits with enhanced scan design). The excellent quality of the achieved results can be seen from further tables in [25, 26, 27] where an extensive comparison to other state-of-the-art tools is made.

	gate–delay		non-robust seq.		robust seq.	
circuit	tested	time [s]	# tested	time [s]	# tested	time [s]
c432	774	0.6		—		
c499	910	1.0		—		
s713	829	0.4	4394	1.9	831	1.1
s991	1474	0.8	10590	3.8	2188	3.2
c2670	4151	9.2		—		
c3540	5459	12.9		—		
c5315	8780	10.5		—		
s5378	6795	13.0	19044	44.2	16218	269.1
c6288	12427	20.9		—		
c7552	12141	30.5		—		
s9234	10698	78.6	33397	1500.1	8454	2085.3
s13207	15323	525.6	129297	4290.4	6122	617.1
s15850	18397	138.3	355325	62853.7	16173	1383.6
s35932	55806	335.5	49784	8089.2	13504	11442.2
s38417	49388	612.6	991057	19940.6	204491	80663.8
s38584	58675	487.1	140609	8281.3	25180	4133.2

Table 3: Result of test pattern generation part 1

Results for optimization of mapped netlists with respect to delay are provided in Table 5. The basic idea and the approach, that applies our implication engine to verify the permissibility of circuit transformations, is described in [6]. Each circuit of Table 5 is optimized by SIS using "script_rugged". The number of gates, literals, and the circuit delay before and after optimization as well as the required CPU time are given. Remarkably, 8.1% of the gates and 3.1% of the literals have been removed resulting in an average delay reduction of 18.8%.

Results for equivalence checking of netlists are presented in Table 6. It lists the total time required for equivalence checking, i.e. ATPG plus computation of indirect implications, and the time consumed by the latter in columns 2 and 3, respectively. The maximal depth of AND-OR search necessary for successful verification is also given in column 4. We provide these early results in order to show that our implication engine forms an excellent data structure for building an efficient equivalence

circuit	non–robust		robust		stuck–at	
	# tested	time [s]	# tested	time [s]	# tested	time [s]
c432	15855	9.6	3730	31.9	520	0.2
c499	367744	112.5	133557	17242.4	750	0.1
c2670	130626	53.9	15370	60.3	2630	1.0
c3540	1202580	6032.3	88354	10595.7	3291	2.4
c5315	342117	643.4	81435	5251.8	5291	1.2
s5378	21928	11.2	18656	43.6	4397	1.3
c6288	30688133	61832.2	26254	31225.4	7710	0.6
c7552	277244	1499.4	86252	5746.0	7419	5.2
s9234	59854	50.6	21389	153.4	6475	18.2
s13207	476145	1364.4	27603	848.5	9608	15.6
s15850	10782994	21320.7	182673	2221.8	11330	9.0
s38417	1138194	2385.6	598062	13001.5	30859	68.6
s38584	334927	2004.1	92239	2071.5	34493	88.7

Table 4: Result of test pattern generation part2

circuit	# gates		# literals		delay		time
	before	after	before	after	before	after	[s]
c432	150	140	318	302	29.9	26.4	238
c499	370	352	920	772	23.4	19.0	2416
c1355	370	352	920	772	23.4	19.0	2400
c880	337	289	722	650	50.6	41.0	658
c1908	488	402	933	803	41.2	33.9	1364
c5315	1576	1374	3249	2790	37.3	31.0	16288
c6288	3148	3009	5357	5923	117.7	92.3	60083
$\sum$:	6439	5918	12419	12012	323.5	262.6	-
red.:		8.1%		3.2%		18.8%	-

Table 5: Results of delay optimization

checker. Therefore, our straightforward approach adopts the basic idea of the well-known equivalence checker HANNIBAL [29] but does not include its advanced heuristics, e.g. observability implications and heuristics for candidate selection. Nevertheless, the results shown in Table 6 are comparable to the ones reported in [29]. This shows that our implication engine is well suited for equivalence checking. Please note, that it is easily incorporated into state-of-the-art implication based or hybrid, i.e. BDDs combined with implications, equivalence checkers such that these approaches can benefit, too.

circuit	time		level_{max}
	total	indirect implications	
c432	1.3s	1.2s	1
c499	4.0s	3.7s	1
c1355	16.9s	16.4s	1
c1908	00:01:08	00:01:04	1
c2670	34.9s	18.1s	2
c3540	00:24:03	00:12:49	2
c5315	00:03:17	00:02:38	1
c6288	8.9s	4.4s	1
c7552	00:09:50	00:08:25	3

Table 6: Results for verifying against redundancy free circuits

6 Conclusion

In this paper we have proposed an efficient implication engine working on a flexible data structure called implication graph. It has been shown that indirect implications can be efficiently computed by analysis of the graph. Experimental results confirm the efficiency and flexibility of our approach.

In the future, our preliminary equivalence checker will be extended by deriving observability implications directly on the implication graph. Furthermore, we will investigate how a hybrid technique using BDDs and the implication graph can be advantageous for equivalence checking.

Acknowledgements

The authors like to thank Bernhard Rohfleisch and Hannes Wittmann for using the implication engine in his netlist optimization tool and developing the path delay ATPG tool, respectively.

A Arbitrary logic

Obviously, there are many possibilities to encode a given arbitrary logic $\mathcal{L}$ which may be more or less suited for performing implications on the resulting implication graph. In general, at least $\lceil \log_2 |\mathcal{L}| \rceil$ bits are necessary to encode a logic value $l \in \mathcal{L}$. Experiments, however, show that an encoding based on the notion of properties is more adequate in our framework. In this scheme, a logic value l is uniquely represented by a set of properties which it either satisfies or not. Since $\mathcal{L}$ may contain composite logic values every property p requires two bits c_{l_p} and $c^*_{l_p}$ for its encoding as shown in Table 7. This style of encoding requires $2 \cdot |P|$ bits with P denoting the set of properties needed to

encoding		interpretation
c_{l_p}	$c^*_{l_p}$	
0	1	l does not satisfy p
1	0	l satisfies p
0	0	l may satisfy p
1	1	conflict

Table 7: Property based encoding of logic values

uniquely encode $\mathcal{L}$.

For example, while property *“l equals logical 1”* is sufficient for encoding $\mathcal{L}_3$ (cf. Table 1), the encoding of logic $\mathcal{L}_{10}$ given in Table 8 is based on the following three properties:

Property 1: true if final value is equal to logical 1, false otherwise.

Property 2: true if signal value is 0*s*, false if signal cannot become 0s.

Property 3: true if signal value is 1*s*, false if signal cannot become 1s.

In case of property based encoding Definition 1 can easily be generalized.

$x \in \mathcal{L}_{10}$	encoding					
	c_{x_v}	$c^*_{x_v}$	c_{x_0}	$c^*_{x_0}$	c_{x_1}	$c^*_{x_1}$
X	0	0	0	0	0	0
$U0$	0	0	0	0	0	1
$U1$	0	0	0	1	0	0
U	0	0	0	1	0	1
$0x$	0	1	0	0	0	1
$0\overline{s}$	0	1	0	1	0	1
$0s$	0	1	1	0	0	1
$1x$	1	0	0	1	0	0
$1\overline{s}$	1	0	0	1	0	1
$1s$	1	0	0	1	1	0

Table 8: 10-valued logic and its encoding

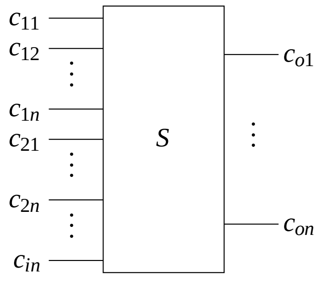
DEFINITION 6 *An assignment is called non-conflicting iff $c_{x_p} \wedge c^*_{x_p} \Leftrightarrow 0$ holds for all signal variables x and all properties p needed for encoding.*

As some assignments to encoding bits do not correspond to valid encoded logic values, so-called logic inherent clauses have to be added to the clause database (or the implication graph). These clauses guarantee that such invalid assignments do not satisfy the *CNF* description of the circuit. They need to be computed only once for a given logic and encoding. For example, the encoding of $\mathcal{L}_{10}$ yields six binary logic inherent clauses while $\mathcal{L}_3$ does not require any.

The truth tables of the supported modules with respect to $\mathcal{L}$ are converted to their corresponding encoded tables according to the principle scheme of Figure 11. In general, an i-input module im-

Figure 11 Mappings due to encoding and corresponding circuit S

$$\begin{tikzcd}
(l_1,\ldots,l_i) \in \mathcal{L}^i \arrow[r, "\mathcal{B}"] \arrow[d, dashed, bend right, "\mathcal{A}"'] \arrow[d, dashed, "\mathcal{A}"] \arrow[d, dashed, bend left, "\mathcal{A}"] \arrow[d, "\underline{\mathcal{A}}"] & l_o \in \mathcal{L} \\
(\underline{c}_1,\ldots,\underline{c}_i) \in \{0,1\}^{i\cdot n} \arrow[r, "\mathcal{C}"] & \underline{c}_o \in \{0,1\}^n \arrow[u, "\mathcal{A}^{-1}"]
\end{tikzcd}$$



plements a mapping $\mathcal{B}$ from $\mathcal{L}^i$ to $\mathcal{L}$. As every logic value l is encoded by n encoding bits, we are interested in a mapping $\mathcal{C}$ from the space of encoded inputs $\underline{c}_I \in \{0,1\}^{i\cdot n}$ to the space of encoded outputs $\underline{c}_o \in \{0,1\}^n$. Given the mapping $\mathcal{A}$ from space $\mathcal{L}$ to the corresponding encoded representation $\underline{c} \in \{0,1\}^n$, mapping $\mathcal{C}$ computes as $\mathcal{C} = \underline{\mathcal{A}}^{-1} \circ \mathcal{B} \circ \mathcal{A}$. Interestingly, mapping $\mathcal{C}$ can be interpreted as representing a combinational circuit S having $i \cdot n$ inputs and n outputs as shown in Figure 11. Similar to the AND-gate of Table 2, circuit S can be optimized exploiting the don't cares specified by the requirement of non-conflicting assignments. For example, the optimized table of an AND-gate in $\mathcal{L}_{10}$ contains six binary and three ternary clauses. As shown in Section 2, the corresponding implication subgraph is deduced from the optimized table and added to the module database.

References

[1] W. Kunz and P. R. Menon, "Multi-level logic optimization by implication analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 6–13, November 1994.

[2] M. Chatterjee, D. K. Pradhan, and W. Kunz, "LOT: Logic optimization with testability - new transformations using recursive learning," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 318–325, November 1995.

[3] D. Stoffel, W. Kunz, and S. Gerber, "And/or reasoning graphs for determining prime implicants in multi-level combinational networks," in *Asia and South Pacific Design Automation Conference*, pp. 529–538, January 1997.

[4] L. A. Entrena and K.-T. Cheng, "Combinational and sequential logic optimization by redundancy addition and removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 14, pp. 909–916, July 1995.

[5] S.-C. Chang and M. Marek-Sadowska, "Perturb and simplify: Multi-level boolean network optimizer," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 2–5, November 1994.

[6] B. Rohfleisch, B. Wurth, and K. Antreich, "Logic clause analysis for delay optimization," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 668–672, June 1995.

[7] W. Kunz and D. K. Pradhan, "Recursive learning; a new implication technique for efficient solutions to cad problems — test, verification, and optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 13, pp. 1143–1158, September 1994.

[8] J. Jain, R. Mukherjee, and M. Fujita, "Advanced verification techniques based on learning," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 420–426, June 1995.

[9] W. Kunz, D. K. Pradhan, and S. M. Reddy, "A novel framework for logic verification in a synthesis environment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 15, pp. 20–32, January 1996.

[10] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 463–466, June 1996.

[11] D. K. Pradhan, D. Paul, and M. Chatterjee, "Verilat: Verification using logic augmentation and transformations," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 88–95, November 1996.

[12] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "Aquila: An equivalence verifier for large sequential circuits," in *Asia and South Pacific Design Automation Conference*, pp. 455–460, January 1997.

[13] M. H. Schulz, E. Trischler, and T. M. Sarfert, "Socrates: A highly efficient automatic test pattern generation system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 7, pp. 126–137, January 1988.

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

[15] W. Kunz and D. K. Pradhan, "Accelerated dynamic learning for test pattern generation in combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 12, pp. 684–694, May 1993.

[16] R. Mukherjee, J. Jain, and D. Pradhan, "Functional learning: A new approach to learning in digital circuits," in *IEEE VLSI Test Symposium*, pp. 122–127, April 1994.

[17] T. Larrabee, “Test pattern generation using boolean satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 11, pp. 4–15, January 1992.

[18] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Combinational test generation using satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 15, pp. 1167–1176, September 1996.

[19] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, “A transitive closure algorithm for test generation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, vol. 12, pp. 1015–1028, July 1993.

[20] J. P. M. Silva and K. A. Sakallah, “Grasp — a new search algorithm for satisfiability,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 220–227, November 1996.

[21] E. Koutsofios and S. C. North, “Drawing graphs with dot,” Tech. Rep. 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill,NJ, September 1991.

[22] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[23] E. Rich, *Artificial Intelligence*. McGraw-Hill, 1983.

[24] F. Maamari and J. Rajski, “A method of fault simulation based on stem regions,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD*, pp. 212–220, February 1990.

[25] M. Henftling, H. Wittmann, and K. J. Antreich, “A formal non–heuristic atpg approach,” in *European Design Automation Conference with EURO-VHDL (EURO-DAC)*, pp. 248–253, September 1995.

[26] M. Henftling and H. Wittmann, “Bit parallel test pattern generation for path delay faults,” in *European Design and Test Conference (ED&TC)*, pp. 521–525, March 1995.

[27] H. Wittmann and M. Henftling, “Path delay atpg for standard scan designs,” in *European Design Automation Conference with EURO-VHDL (EURO-DAC)*, pp. 202–207, September 1995.

[28] M. Henftling, H. C. Wittmann, and K. J. Antreich, “A single–path–oriented fault–effect propagation in digital circuits considering multiple–path sensitization,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 304–309, November 1995.

[29] W. Kunz, “Hannibal: An efficent tool for logic verification based on recursive learning,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 538–543, 1993.