SAT-Based ATPG Using Multilevel Compatible Don't-Cares

NIKHIL SALUJA
Univerity of Colorado
and
KANUPRIYA GULATI and SUNIL P KHATRI
Texas A & M University

In a typical IC design flow, circuits are optimized using multilevel don't cares. The computed don't cares are discarded before Technology Mapping or Automatic Test Pattern Generation (ATPG). In this paper, we present two combinational ATPG algorithms for combinational designs. These algorithms utilize the multilevel don't cares that are computed for the design during technology independent logic optimization. They are based on Boolean Satisfiability (SAT), and utilize the single stuck-at fault model. Both algorithms make use of the Compatible Observability Don't Cares (CODCs) associated with nodes of the circuit, to speed up the ATPG process. For large circuits, both algorithms make use of approximate CODCs (ACODCs), which we can compute efficiently. Our first technique speeds up fault propagation by modifying the active clauses in the transitive fanout (TFO) of the fault site. In our second technique, we define new *j-active* variables for specific nodes in the transitive fanin (TFI) of the fault site. Using these j-active variables we write additional clauses to speed up fault justification. Experimental results demonstrate that the combination of these techniques (when using CODCs) results in an average reduction of 45% in ATPG runtimes. When ACODCs are used, a speed-up of about 30% is obtained in the ATPG run-times for large designs. We compare our method against a commercial structural ATPG tool as well. Our method is slower for small designs, but for large designs, we obtain a 31% average speedup over the commercial tool.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Design, Performance, Reliabilty

Additional Key Words and Phrases: Automatic test pattern generation (ATPG), Boolean satisfiabilty (SAT), don't cares, testing

ACM Reference Format:

Saluja, N., Gulati, K., and Khatri, S. P. 2008. SAT-based ATPG using multilevel compatible don't-cares. ACM Trans. Des. Autom. Electron. Syst. 13, 2, Article 24 (April 2008), 18 pages, DOI = 10.1145/1344418.1344420 http://doi.acm.org/10.1145/1344418.1344420

Author's address: S. P. Khatri, Department of Electrical and Computer Engineering, 333 WERC, MS 3259, Texas A & M University, College Station, TX, 77843-3259.

1. INTRODUCTION

In order to ensure that a manufactured IC is error-free, IC vendors typically perform a set of tests before shipping each die. Manufacturing defects manifest themselves as logical faults, which are mathematically modeled as circuit nodes becoming statically 1 (stuck-at-1) or 0 (stuck-at-0). Using this *single stuck-at fault model*, automatic test pattern generation (ATPG) algorithms determine a set of tests (vectors on the primary inputs of the circuit) to test all possible stuck-at faults in a design.

In this article we propose two techniques that significantly speed up SAT-based ATPG. Both techniques use the compatible observability don't cares (CODCs) associated with nodes of the circuit. These don't cares are generally computed during technology independent optimization of the circuit and are discarded thereafter. In our approach we save these don't cares and use them to speed up ATPG.

The key contribution of this article is a pair of orthogonal techniques to augment SAT-based ATPG with circuit Don't Care information.

—In our first technique we augment the active clauses of the nodes in the transitive fanout of the fault site, with approximate CODC information to speed up the process of fault propagation.

—In the second technique we define new active variables for specific nodes in the transitive fanin of the fault. Additional clauses which encode the approximate CODCs, using these active variables are then added to speed up the process of fault justification.

This paper can be extended to address sequential ATPG, by unfolding a sequential circuit in time, computing sequential Don't Cares and applying the same techniques as described in the sequel. Further, in such a scenario, sequential don't cares can be used to additionally enhance the technique. The experiments for this paper are conducted on combinational designs.

The rest of this article is organized as follows. In Section 2, we provide definitions that are used in the rest of the paper. Section 3 discusses previous work while Section 4 describes our improved SAT-based ATPG scheme. Experimental results are presented in Section 5, and conclusions are drawn in Section 6.

2. PRELIMINARIES AND TERMINOLOGY

Definition 2.1. The Boolean difference [McCluskey 1986] of a function f with respect to x is defined as

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\bar{x}},$$

where f_x and $f_{\bar{x}}$ are f with x set to 1 and 0 respectively.

Definition 2.2. Given a multilevel combinational Boolean network C, a single stuck-at fault $f = f(x,B)$ causes a node x in C to be permanently stuck at logic value B (where $B \in \{0, 1\}$). The faulty circuit, denoted by C_f is then C with the faulty node x assigned to B.

Definition 2.3. The test for a node x stuck-at-0 is defined as

$$(x) \cdot \left(\frac{\partial f}{\partial x}\right),$$

where f is some primary output of the multilevel Boolean circuit.

In the expression above, the first term x represents the fault excitation and justification conditions and the second term $\frac{\partial f}{\partial x}$ represents the fault propagation condition.

Similarly, the test for a node x stuck-at-1 is defined as

$$(\overline{x}) \cdot \left(\frac{\partial f}{\partial x}\right).$$

Definition 2.4. A conjunctive normal form (CNF) Boolean formula f on n Boolean variables $x_1, x_2, \ldots, x_n$ is a conjunction (logical AND) of m clauses $c_1, c_2, \ldots, c_m$. Each clause c_i is the disjunction (logical OR) of its constituent literals.

For example,

$$f = (x_1 + x_3) \cdot (x_1 + \overline{x_2})$$

is a CNF formula with two clauses, $c_1 = (x_1 + x_3)$ and $c_2 = (x_1 + \overline{x_2})$.

Definition 2.5. The problem of Boolean satisfiability (SAT) is to determine whether a Boolean formula in conjunctive normal form (CNF) has a satisfying assignment.

SAT is an NP complete problem [Garey and Johnson 1979]. Several heuristics exist for efficient solution of SAT. Among these are Zchaff [Moskewicz et al. 2001] and GRASP [Silva and Sakallah 1996]. In GRASP, efficiency results from the use of non-chronological backtrack. Zchaff improves these results further by an efficient mechanism of 'watching' literals in the clauses.

Definition 2.6. The Observability Don't Care of node y_j in a multilevel Boolean network with respect to output z_k is

$$ODC_{jk} = \{x \in B^n \; s.t. \; z_k(x)|_{y_j=0} = z_k(x)|_{y_j=1}\}.$$

Here $B \in \{0, 1\}$.

In other words ODC_{jk} is the set of minterms of the primary inputs for which the value of y_j is *not observable* at z_k [Savoj and Brayton 1990]. This can also be denoted as

$$ODC_{jk} = \left(\overline{\frac{\partial z_k}{\partial y_j}}\right),$$

where

$$\frac{\partial z_k}{\partial y_j} = z_k(x)|_{y_j=0} \oplus z_k(x)|_{y_j=1}. \tag{1}$$

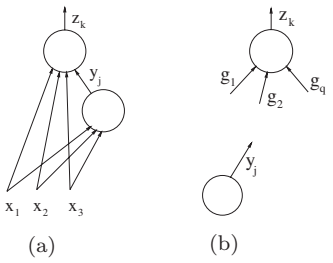

Fig. 1. Don't Care example networks.

In the network of Figure 1(a), z_k explicitly depends on y_j so $\frac{\partial z_k}{\partial y_j}$ can be computed using equation 1. In general, when z_k is not explicitly dependent on y_j, as is the case in Figure 1(b), we can compute $\frac{\partial z_k}{\partial y_j}$ using the chain rule:

$$\begin{aligned}\frac{\partial z_k}{\partial y_j} = {} & \frac{\partial z_k}{\partial g_1}\cdot\frac{\partial g_1}{\partial y_j} \oplus \frac{\partial z_k}{\partial g_2}\cdot\frac{\partial g_2}{\partial y_j} \oplus \cdots \oplus \frac{\partial z_k}{\partial g_q}\cdot\frac{\partial g_q}{\partial y_j} \\ & \oplus \frac{\partial^2 z_k}{\partial g_1 g_2}\cdot\frac{\partial g_1}{\partial y_j}\cdot\frac{\partial g_2}{\partial y_j} \oplus \frac{\partial^2 z_k}{\partial g_1 g_3}\cdot\frac{\partial g_1}{\partial y_j}\cdot\frac{\partial g_3}{\partial y_j} \oplus \cdots \\ & \oplus \frac{\partial^2 z_k}{\partial g_1 g_q}\cdot\frac{\partial g_1}{\partial y_j}\cdot\frac{\partial g_q}{\partial y_j} \oplus \cdots \oplus \frac{\partial^2 z_k}{\partial g_{q-1} g_q}\cdot\frac{\partial g_{q-1}}{\partial y_j}\cdot \\ & \frac{\partial g_q}{\partial y_j} \oplus \frac{\partial^3 z_k}{\partial g_1 g_2 g_3}\cdot\frac{\partial g_1}{\partial y_j}\cdot\frac{\partial g_2}{\partial y_j}\cdot\frac{\partial g_3}{\partial y_j} \oplus \cdots \\ & \oplus \frac{\partial^q z_k}{\partial g_1 g_2 \ldots g_q}\cdot\frac{\partial g_1}{\partial y_j}\cdot\frac{\partial g_2}{\partial y_j}\cdots\frac{\partial g_q}{\partial y_j}. \end{aligned} \tag{2}$$

Once a node function is changed by minimizing [Brayton et al. 1984] it against its ODCs, the ODCs of the other nodes must be recomputed. To avoid recomputation of ODCs during optimization, Compatible Observability Don't Cares (CODCs) [Savoj and Brayton 1990] were developed. The CODC of a node is a subset of the ODC for that node. Unlike ODCs, CODCs have a property that one can *simultaneously* change the function of all nodes in the network as long as each of the modified functions are contained in their respective CODCs.

Definition 2.7. A node x is said to be the dominator of another node y (i.e., node y is dominated by node x), if all paths from y to the primary outputs go through node x [Tarjan 1974].

This is illustrated by means of an example in Figure 2. We can see that all paths from nodes c, e, and g pass through node x. Hence node x dominates nodes c, e, and g.

In the rest of this section, we briefly review SAT-based ATPG, with a view to providing a framework for discussing our approach.

In the SAT-based ATPG method we first generate a formula (in conjunctive normal form (CNF)) to represent the test for the fault. Every gate of the circuit has a CNF formula associated with it which represents the function performed by the gate. This formula is true iff the variables representing the gate's inputs

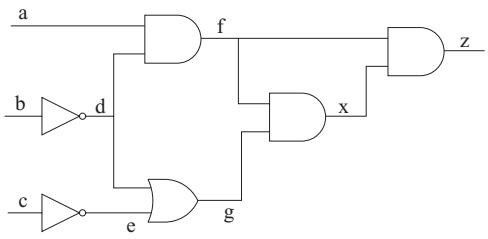

Fig. 2. Example circuit.

and output take on values consistent with its truth table. For example, consider a 2-input AND gate with x and y as inputs and z as output. The CNF formula for the AND gate is written as:

$$(\overline{z} + x) \cdot (\overline{z} + y) \cdot (z + \overline{x} + \overline{y}).$$

A CNF formula for the entire circuit is obtained by forming the conjunction of the CNF formulas of all gates of the circuit. This CNF formula describes the *good* (fault-free) circuit behavior. The faulty circuit is a copy of the fault-free circuit with new *faulty* variables for the gates affected by the fault (i.e., gates in the transitive fanout of the fault). A CNF formula describing the faulty circuit is obtained in a similar manner as the formula obtained for the good circuit. A clause for the faulty circuit is written only if it is different from the good circuit clauses. Next, fault detection clauses are written. These clauses consist of:

—Clauses representing the XOR of each good circuit output and the corresponding faulty circuit output.
—Clauses representing the logical OR of each XOR output above.

For SAT-based ATPG, a CNF formula for the ATPG instance is constructed by taking the conjunction of the good circuit clauses (conjunction of good circuit clauses of each gate), the faulty circuit clauses (conjunction of faulty clauses for each gate in the TFO of the fault site) and fault detection clauses. This CNF formula is then solved using a SAT solver. If a satisfying assignment S exists, then the fault is testable else it is redundant. The assignment of values to primary input variables in S represents the test vector for the fault. The entire process is repeated for each distinct fault in the circuit.

To speed up the process of finding a satisfying assignment, *active clauses* [Larrabee 1992] are added to the composite CNF formula. These are described in Section 4.1.

Recall that our technique proposes to use CODCs that are generated during the technology independent optimization. After mapping, several technology independent nodes may get collapsed into a gate. Finding the CODCs of gates in this context is simple. Suppose we map a set of 3 technology independent nodes a, b and c into a gate G. Suppose b and c are fanouts of a. Then the CODC of G can be computed as the composition of the CODCs of b and c in the CODC of a.

3. PREVIOUS WORK

In the past, the ATPG problem has received extensive attention in academia and industry. ATPG techniques can be classified as *structural*, *algebraic*, and *hybrid*.

A large fraction of ATPG techniques use structural methods. The D-Algorithm [Roth 1966] was one of the earliest known ATPG techniques. This method tried to perform fault justification and propagation by a structural search on *all* nodes of a circuit. This was improved by PODEM [Goel 1981] where the search space was restricted to the primary inputs of the circuit, resulting in a significantly more efficient ATPG technique. Techniques like FAN [Fujiwara and Shimono 1983] further improved performance by exploiting immediately applicable implications, headlines, and multiple back-traces. An algorithm that exploits the notion of circuit *dominators* was introduced in Kirkland and Mercer [1988]. Structural ATPG techniques based on Boolean learning were introduced in Schulz et al. [1988] [Kunz and Pradhan 1994]. These techniques augmented the structure-based search process by performing additional static or dynamic learning of logical implications in the circuit.

Algebraic techniques are elegant from a mathematical perspective, and involve algebraic manipulation of the equations describing the testability condition. The most well known of these techniques is [Sellers et al. 1968]. In general, these techniques can prove to be expensive and therefore there has not been much attention devoted to them.

Hybrid techniques are more recent, and they typically utilize a mixed structural and functional approach.

One such hybrid technique is the SAT-based ATPG technique introduced by Larrabee [1992] and explored further in TEGUS [Stephan et al. 1996]. These techniques translate the testability condition into a Boolean Satisfiability (SAT) instance, which retains the circuit structure. A test for the circuit is now obtained by invoking a SAT solver. During this step, the circuit structure is not explicitly used in determining a test. SAT-based ATPG techniques were shown to be robust and fast, and our algorithms are developed in a SAT-based ATPG framework.

Shi et al. [2005] exploit advanced SAT techniques for ATPG. They evaluate the performance of different SAT solvers in ATPG, and show the potential for problem specific heuristics for speeding up SAT. Further, they reduce the time taken to generate the CNF by efficient memory allocation. These approaches for speeding up CNF generation and SAT solving for ATPG are orthogonal to our scheme (since they don't enhance the ATPG process by using Don't Cares) and hence can be easily combined with our current approach for further runtime improvements.

Another SAT-based technique was reported in Tafertshofer et al. [1997]. In this approach, the authors perform justification and propagation on an implication graph (IG) structure. This represents an efficient implementation of a SAT-based method to analyze Boolean networks. Our techniques are orthogonal to those of Tafertshofer et al. [1997]. It would be interesting to see the performance of a method which combines our approach and that of Tafertshofer et al. [1997].

Gupta et al. [2001] present a dynamic method to detect and remove inactive clauses during SAT. Their approach is orthogonal to ours, and it would be an interesting research problem to see how a combined approach performs. Our method would make the search of the remaining clauses faster since it augments the clauses with CODC information.

The efforts of Zhaohui et al. [2005], Velev [2004], and Safarpour et al. [2004] are similarly motivated. In Safarpour et al. [2004], SAT is sped up for a circuit instance by labeling variables as *lazy* when they are determined to be noncontrolling. For example, if a logic cone feeds into an AND gate, one of whose inputs is a 0, then all the variables in the logic cone can be disregarded by the SAT solver. The approach of Zhaohui et al. [2005] uses don't care literals, which are treated differently during the solution process. A similar approach is reported in Velev [2004], where unobservable literals are added to each clause. These three approaches are quite similar in their motivation. The main difference between these approaches and ours is that our computed don't cares do not need to be updated during a SAT run, since we utilize Compatible Observability Don't Cares (CODCs). Further, our approach is different from those of Zhaohui et al. [2005], Velev [2004], and Safarpour et al. [2004], since the information used by these approaches does not utilize CODCs, but rather the structure of the circuit.

Radecka and Zilic [2001, 2002] describe techniques to identify redundant gate and wire replacement conditions. The work is based on SAT, and the authors utilize CODCs (which they refer to as approximate ODCs). The difference of our approach from Radecka and Zilic [2001, 2002] lies in the fact that we utilize *approximate* CODCs [Saluja and Khatri 2004] (in order to handle large designs for which CODCs cannot be computed). The use of approximate CODCs makes our technique extremely robust, allowing it to handle arbitrarily large designs. The application setting of our paper is quite different from Radecka and Zilic [2001, 2002], and hence our technical approach is also different. Further, their experiments are performed on smaller examples, and not compared to a commercial tool (unlike our experiments).

SPIRIT is a SAT-based ATPG tool [Gizdarski and Fujiwara 2000] that implements structural concepts like unjustified lines and static learning. An extended version of these heuristics are presented by the same authors in Gizdarski and Fujiwara [2002]. However, unlike our method, the learning applied is local. Our method implicitly utilizes dynamic learning since the underlying SAT solver incorporates this. Our method also utilizes structural information (in the form of CODCs) to make the SAT-based search more efficient.

Bhattacharya et al. [1995] present an ROBDD-based ATPG tool. In this hybrid technique, circuit structure is lost when the ROBDD of the circuit is constructed, but structural information is used to guide the ROBDD-based test generation process.

There have been prior efforts to use ATPG in order to improve the speed or quality of logic synthesis and verification [Chang and Marek-Sadowska 2002; Chang and Marek-Sadowska 1994; Chang et al. 1996; Kunz 1994; Huang et al. 2001; Huang et al. 2000]. The guiding principle of these papers is exactly the reverse of our article (which uses logic synthesis information in the form of multilevel don't cares) to speed up ATPG.

4. OUR APPROACH

Our approach utilizes a SAT-based formulation of ATPG. We utilize the efficient SAT solver Zchaff [Moskewicz et al. 2001] to solve the SAT instance that arises from the ATPG problem. Our approach has two orthogonal parts.

In the first part we augment the active clauses [Larrabee 1992] in the transitive fanout of the fault by utilizing the available CODC information. This improves the efficiency of fault propagation. In the second part we introduce new active variables and new clauses for selected nodes in the TFI of the fault. These clauses utilize CODC information as well, and help improve the efficiency of fault justification.

4.1 Efficient Fault Propagation

In Larrabee's SAT-based approach [Larrabee 1992], fault propagation is sped up by adding *active clauses* to the composite CNF formula representing the testability condition. If a fault is testable, then there must be at least one path from the fault site to the primary output, such that every node along that path has different good and faulty values. Hence, every node in the transitive fanout of the fault is allocated an *active variable*. A node x is active (represented as x_a) if its value in the good circuit is different from its value in the faulty circuit. The clauses for this condition are:

$$x_a \Rightarrow x \oplus x_f .$$

If a node is active then at least one of its fanouts must be active. The clauses for this condition are obtained from the expression:

$$x_a \Rightarrow \sum_{y \in FO(x)} (y_a).$$

Finally, one of the outputs must be active.

We write active clauses for every node in the TFO of the fault, in order to guide the SAT solver to search in the relevant region of the circuit. Since the SAT solver natively has no notion of circuit structure, such guidance is quite essential. By adding active variables and the associated active clauses, we effectively incorporate some structural information in the SAT search process. Correctness is not compromised if some or all active variables and/or active clauses are omitted.

As an example consider Figure 2. Assume that node x is the fault node. If x_a and z_a are the active variables for nodes x and z respectively, then the active clauses would be written as

$$(\overline{x_a} + x + x_f) \cdot (\overline{x_a} + \overline{x} + \overline{x_f})$$

and

$$(\overline{z_a} + z + z_f) \cdot (\overline{z_a} + \overline{z} + \overline{z_f}).$$

Along with these clauses, the other active clauses that are added are $(\overline{x_a} + z_a)$ and (z_a).

In our method, we add new clauses to guide fault propagation. These clauses make use of the Compatible Observability Don't Cares (CODCs) of the nodes

in the transitive fanout of the fault site. It is generally the case that these don't cares are computed during technology independent optimization of a digital circuit so there is no computational overhead in adding the proposed clauses.

We could use full ODCs as well, and these would result in improved results. However, full ODCs are harder to compute. Since CODCs are typically computed during the technology independent phase of a design, we simply re-utilize them in our approach.

CLAIM 1. *If the assignment of the inputs of a node n is contained in the CODC of n, then n cannot be on a path consisting of active nodes starting from the fault site and ending at a primary output.*

The above claim can be proved easily since the CODC of a node is contained in its ODC. In our approach, we assert that if the input assignment of a node is contained in its CODC, the node cannot be active, thereby avoiding unnecessary search. Hence, for node x in Figure 2, the new clauses that can be added to the circuit are

$$(x_a \Rightarrow \overline{CODC_x});$$

that is,

$$(\overline{x_a} + \overline{CODC_x}).$$

In general, the clauses generated by the above expression may have large number of literals. In our implementation, we ensure that the resulting clauses have at most k literals, by deleting clauses with $k + 1$ or more literals. Here k is a user-specified parameter. As a result, we actually implement

$$(x_a \Rightarrow \overline{S_x}),$$

where

$$\overline{S_x} \supseteq \overline{CODC_x}.$$

These clauses are added for all nodes in the transitive fanout of the fault. Experimental results show an average improvement of 41.5% in the SAT runtime due to the addition of these new clauses.

4.2 Efficient Fault Justification

We now describe our method of adding new *j-active* clauses for selected nodes in the TFI of the fault site. These new clauses help speed up fault justification. These clauses can be expressed in terms of the ODC of the nodes in the TFI of the fault. However, since the CODCs of every node have already been computed, we make use of them instead. To do this we first find all the nodes in the TFI of the fault (except for primary inputs) for which the fault node is a *dominator*. We assign j-active variables[1] to all such nodes. Because these new j-active nodes are dominated by the fault node, we can reduce the chain rule (Equation (2) in

[1]For a node p, we denote its j-active variable as p_{act}. This notation is intentionally different from the notation used for active variables corresponding to nodes in the TFO of the fault site.

Section 2) to:

$$\frac{\partial z_k}{\partial y_j} = \left(\frac{\partial z_k}{\partial g_1}\right) \cdot \left(\frac{\partial g_1}{\partial y_j}\right), \tag{3}$$

where z_k is a primary output, g_1 is the fault node and y_j is a node which is dominated by g_1. This is valid since $\frac{\partial g_n}{\partial y_j} = 0$ for $n > 1$. From this equation, we note that the first term of the right hand side (RHS) is the care-set associated with fault propagation. The second term of the RHS is the care-set associated with fault justification. Since we require that the fault be justified *and* propagated, we need to ensure that the care points used for the node y_j satisfy both terms of the RHS. In other words, the care points used for the node y_j must be the care points obtained by computing the Boolean difference of the output z_k with respect to y_j (the condition on the left hand side (LHS) of Equation (3)).

For a multi-output circuit, we would need to compute the care points for node y_j as the complement of ODC_j, where

$$ODC_j = \prod_{k \in PO} (ODC_{jk}).$$

In practice, we compute the care points for node y_j as the complement of $CODC_j$, since these have already been computed during technology-independent logic optimization.

The new j-active clauses for a node p which is dominated by the fault site are written in two parts:

(1) A node p is said to be j-active if the immediate fanins of its fanout nodes (other than the node p itself) do not determine the value of the fanout node.
(2) If a node is j-active then its input assignment must not be contained in its CODC. In other words, for node p which is dominated by the fault site, we write clauses:

$$p_{act} \Rightarrow \overline{CODC_p}.$$

The traversal of the circuit to write the first part of the j-active clauses is illustrated by the following algorithm. Note that we run this algorithm on each *node* (in the transitive fanin of the fault site) which is dominated by the fault site.

```
Active_clauses (node, fault_site) {
   ForEach_Fanout (node) {
      AddActive_clause (node)
      if (fanout = fault_site)
         return
      else
         Active_clauses (fanout, fault_site)
   }
   return
}
```

The function *AddActive_clause*(*node*) adds the first part of the new j-active clauses based on the type of gate being implemented at the node.

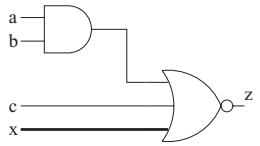

Fig. 3. AOI-211 gate.

Next, we write the second part of the new j-active clauses.

$$p_{act} \Rightarrow \frac{\partial g_1}{\partial p}.$$

For the AOI-211 gate of Figure 3 the first part of the j-active clauses that would be added for node x would be

$$(\overline{c} \cdot \overline{a \cdot b}) \Rightarrow x_{act}.$$

This is because the value of z is determined by x only if $c = 0$ and $a \cdot b = 0$.

The second part of the j-active clauses would be

$$(x_{act} \Rightarrow \overline{CODC_x}).$$

The addition of these new active clauses for our working example in Figure 2 is described next.

Assume node x is the fault node. Here the nodes c, e, and g are dominated by x. Now, node g will determine the value of x only if f equals 1. Hence the new active clauses for node g can be written as

$$(f \Rightarrow g_{act}) \cdot (g_{act} \Rightarrow \overline{CODC_g})$$

i.e

$$(\overline{f} + g_{act}) \cdot (\overline{g_{act}} + \overline{CODC_g})$$

Similarly, we can also write new active clauses for the node e as

$$(\overline{d} \cdot f \Rightarrow e_{act}) \cdot (e_{act} \Rightarrow \overline{CODC_e})$$

i.e

$$(d + \overline{f} + e_{act}) \cdot (\overline{e_{act}} + \overline{CODC_e})$$

Note that j-active clauses for node c are not written since it is a primary input. In this manner we can write new j-active clauses for all nodes in the TFI of the fault node, which are dominated by the fault node. Experimental results using CODCs show an average improvement of 45% in the SAT run times due to the addition of the clauses implemented in Sections 4.1 and 4.2. The incremental improvement obtained by including clauses of this section is only about 5%, since we are able to write new j-active clauses for a small subset of nodes in the TFI of the fault site.

In a traditional SAT-based ATPG flow, active clauses [Larrabee 1992] are utilized. Therefore, implementing our method of Section 4.1 incurs no variable overhead. Also, the number of dominators we found in typical circuits was very small. As a result, we do not report results for the method of Section 4.2 applied in isolation.

4.3 Approximate CODCs

CODCs are computed using an ROBDD [Bryant 1986] based computation. Therefore, it is not possible to compute them for larger designs. Hence, for larger designs, we implement a technique to compute approximate CODCs (ACODCs) [Saluja and Khatri 2004], which computes a large subset of the CODCs quickly. ACODCs can be computed on average 25× faster than CODCs, with an average 33× reduction in memory utilization. This method is robust in that it can be applied to large designs for which the CODC computation does not complete.

We demonstrate the utility of our techniques using CODCs (on circuits for which CODCs can be computed) as well as ACODCs. For larger designs, we utilize ACODCs exclusively. In this case, we simply replace the CODC terms in the clauses described in Sections 4.1 and 4.2 by the ACODCs.

5. EXPERIMENTAL RESULTS

Both our techniques are implemented in SIS [Sentovich et al. 1992]. For our experiments we use the *mcnc91* and *itc99* benchmark circuits. Our experimental procedure consists of reading in a design and running *script.rugged* on the design. This script computes CODCs for the circuit during circuit optimization. For ACODC tests, we replace *full_simplify* with our ACODC [Saluja and Khatri 2004] version of this code. These don't cares are generally computed during technology independent optimization of the circuit and are discarded thereafter. In our approach we save these don't cares and use them to speed up ATPG. Hence the use of these don't cares for ATPG incurs no extra runtime cost. Next we technology-map the circuit using the library *lib2.genlib*. Two mapping schemes are employed, one which attempts to minimize area and another which attempts to minimize delay. Now for each uncollapsed fault in the design, we generate SAT clauses to test the fault, and then invoke Zchaff [Moskewicz et al. 2001] to find a test. Our method is compared with SAT-based ATPG (without don't cares)[2] as well as a commercial ATPG tool. Comparisons with the old method are performed on an IBM IntelliStation running Linux with a 1.7 GHz Pentium-4 CPU and 1 GB of RAM. Comparisons with the commercial tool are run on a Sun Ultra-4 SPARC machine, running SunOS 5.7 (we ran our method *and* the commercial ATPG tool on the same Sun machine). We use the latest version of this commercial tool. The licensing agreement for this tool requires that we do not mention the name of the tool in this paper. In all experiments, *no* random vector simulation is performed and *all* faults are tested using our deterministic procedure. This ensures that runtime comparisons are fair and objective. The reason for this choice is that if random vector simulation is performed, the commercial tool and our tool may test a different set of faults during random vector simulation, making it impossible to draw objective conclusions from the results. All runtimes are in seconds.

[2]The original SIS ATPG algorithm uses a SAT solver internal to SIS. This has been modified such that the SIS ATPG algorithm uses Zchaff [Moskewicz et al. 2001] as the SAT solver. This method is referred to as the 'old method' in this paper. It utilizes the notion of active clauses [Larrabee 1992], but uses no Don't Care enhanced clauses like our method does.

Table I. Average Speed-Up of Our Techniques

DC set	Mapping	% improvement (1)	% improvement (1 + 2)
sub-fanin	*area*	40.86%	45.47%
level	*area*	41.92%	45.93%
all	*area*	41.52%	46.60%
sub-fanin	*delay*	38.71%	45.66%
level	*delay*	42.18%	46.21%
all	*delay*	43.42%	46.21%

Table II. Clause and Variable Overheads for Our Techniques—Medium Sized Circuits

					using CODCs				using ACODCs		
circuit	flts	test	red	old clause	cl % (1)	cl %(1+2)	old vars	var % (2)	cl % (1)	cl %(1+2)	var % (2)
*alu*2	976	968	8	1297*k*	5.12	5.73	411529	0.20	4.96	5.21	0.15
*alu*4	1833	1812	21	4191*k*	4.60	4.87	1366145	0.09	4.32	4.55	0.08
*apex*6	2153	2153	0	497*k*	4.46	5.08	202319	0.17	4.11	4.67	0.15
*apex*7	650	649	1	133*k*	3.23	3.50	57703	0.24	3.01	3.22	0.20
*C*1355	1337	1337	0	2643*k*	4.17	4.62	892533	0.03	3.83	3.99	0.02
*C*1908	1284	1282	2	2414*k*	4.84	5.04	857028	0.06	4.33	4.65	0.05
*C*2670	2304	2297	7	3145*k*	1.44	2.06	1260296	0.17	1.32	1.57	0.15
*C*499	1337	1337	0	2646*k*	3.90	4.09	981219	0.06	3.61	3.89	0.05
*C*880	1216	1216	0	824*k*	1.48	1.83	315654	0.11	1.22	1.45	0.10
*frg*2	2219	2213	6	780*k*	3.48	3.70	323958	0.06	3.18	3.35	0.05
*i*5	844	844	0	147*k*	0.23	0.32	70584	0.09	0.20	0.30	0.08
*i*6	1529	1529	0	195*k*	5.38	5.83	76996	0.15	4.56	4.92	0.12
*i*7	2096	2096	0	298*k*	5.26	5.37	121723	0.12	5.02	5.10	0.09
rot	1954	1953	1	1166*k*	1.34	1.50	479787	0.04	1.11	1.26	0.03
*term*1	498	494	4	131*k*	2.98	4.36	51436	0.32	2.77	3.43	0.22
too_large	810	803	7	518*k*	1.26	1.91	198527	0.20	0.99	1.34	0.15
vda	1092	1092	0	1127*k*	6.82	6.89	383695	0.02	5.96	6.01	0.02
*x*1	862	862	0	198*k*	0.52	1.03	85480	0.14	0.42	0.86	0.10
*x*3	2303	2303	0	508*k*	3.30	3.84	203281	0.15	3.03	3.26	0.11
*x*4	1126	1126	0	211*k*	4.40	4.60	90763	0.16	4.12	4.35	0.13
AVG	—	—	—	—	3.71	4.07	—	0.10	3.36	3.65	0.08

In our experiments, we use three different sets of CODCs (which are implemented in SIS [Sentovich et al. 1992]). Assume that the CODCs of a node n are being computed. These three sets are discussed below.

—The *sub-fanin* set: This set consists of fanin don't cares only for nodes with the same or subset support as the node n.

—The *level* set: This set consists of fanin don't cares only for nodes with the same or subset support as the node n, which have level less than the node n.

—The all set: This set consists of all the don't cares that can be generated for each node.

Table I summarizes the average improvements in SAT runtimes for different don't care sets. Column 1 lists the don't care sets used while Column 2 describes the type of mapping. Column 3 lists the percentage improvement in the SAT runtimes for our first technique over the SIS ATPG algorithm, while Column 4 does the same comparison for both our techniques applied simultaneously. The average improvement in the SAT run-times over the original ATPG method is 41.5% for the first technique and 46% for both techniques applied simultaneously.

Table II describes the clause and variable overhead of our proposed techniques applied to medium sized circuits. Column 1 lists the circuit name,

Table III. Clause and Variable Overheads, and Runtime for Our Techniques—Large Designs

circuit	faults	tested	old clauses	cls. ovh % (1)	cls. ovh % (1+2)	old vars	vars %(2)	norm. time (1)	norm. time (1+2)	time our/comm.	aborts our/comm.
b14_C	31055	30850	286.9*M*	2.11	2.21	98.1*M*	0.06	0.65	0.62	4818.06/5625.34 = 0.85	0/1
b15_C	28950	28037	444.9*M*	2.23	2.34	151.1*M*	0.03	0.83	0.81	14591.42/22370.30 = 0.65	0/0
b17_C	101143	98685	1381.3*M*	1.64	1.77	467.6*M*	0.02	0.72	0.69	36459.48/56912.03 = 0.64	0/2
b20_C	63127	62718	812.6*M*	2.43	2.71	274.0*M*	0.06	0.76	0.74	15534.91/31270.12 = 0.49	0/3
b21_C	64465	64003	851.7*M*	1.28	1.35	287.9*M*	0.02	0.63	0.59	15506.17/38152.79 = 0.40	0/3
b22_C	93309	92838	1171.7*M*	1.32	1.45	396.3*M*	0.02	0.69	0.66	23419.98/6429.08 = 3.64	0/3
AVG	—	—	—	1.835	1.97	—	0.035	0.71	0.685	110330.02/160759.66 = 0.69	—

column 2 reports the number of faults to be tested and columns 3 and 4 report the number of tested and redundant faults respectively. Column 5 lists the total number of clauses (for all the faults) in the old method, while column 6 lists the clause overhead using our first technique (as described in Section 4.1). Column 7 lists the clause overhead using both our techniques (as described in Sections 4.1 and 4.2) simultaneously. Column 8 lists the total number of variables in the old method (over all tested faults), whereas column 9 shows the variable overhead in the new method (new variables are added only in our second technique). Columns 6, 7, and 9 correspond to the use of CODCs. Columns 10, 11, and 12 respectively represent the same overheads as columns 6, 7, and 9 for the case when ACODCs are used in the ATPG computation.

Note that the average clause overhead of our techniques (using CODCs) is low (approximately 4%). The average variable overhead of our second technique is also low (0.1%). The percentage of nodes, on average, that are dominated by fault nodes is about 5%. *Also, note that for our method, there are no aborted faults*. When ACODCs are used, these overheads reduce marginally. In all cases the clause and variable overheads are extremely reasonable.

Table III describes the ATPG results for large designs (for which CODCs cannot be computed). Therefore we use ACODCs for both our algorithms. Columns 1 through 4 of this table are self-explanatory. Columns 5 and 6 respectively represent the clause overhead for our first method in isolation and for both methods together. Column 8 represents the variable overhead for our second method. Columns 9 and 10 represent the runtimes of our techniques using the method of Section 4.1 and using the combination of the methods of Section 4.1 and Section 4.2, both normalized with respect to the old method. We see that for large designs, our techniques deliver about 31.5% speedup compared to the old method (i.e., compared to the use of SAT-based ATPG with just the use of traditional active clauses, but no don't care enhancements). For these large designs, the percentage of nodes, on average, that are dominated by fault nodes is about 3.8%. Column 11 reports the runtime of our method, the runtime of the commercial tool, and the ratio of the two. Also, Column 12 reports the aborted faults of our method and the commercial tool. Note that on average, for these large designs, our method runs 31% faster than the commercial tool. Also, *our method aborts on no faults, while the commercial tool aborts on a handful of faults in some of these designs*. The increased runtime for *b22_C* is attributed to the presence of some hard faults, which our method tested but the commercial tool aborted on.

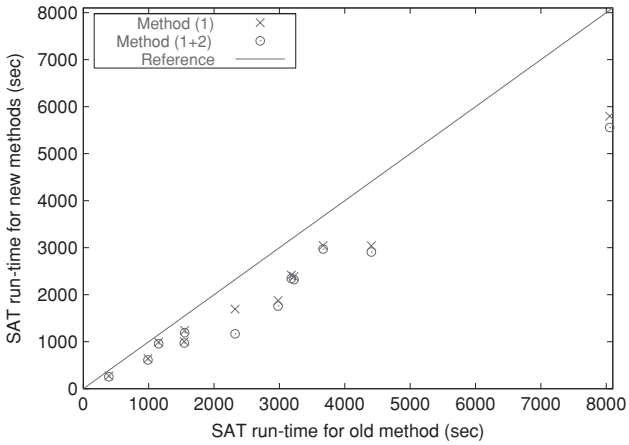

Fig. 4. SAT runtime comparison using large examples.

Table IV. Effect of Limiting Clause Sizes in Our Techniques

	cls.ovh.w/CODC				time w/CODCs				time w/ACODCs	time w/CODCs ($k > 5$)
circuit	$k = 3$	$k = 4$	$k = 5$	$k > 5$	$k = 3$	$k = 4$	$k = 5$	$k > 5$	$k > 5$	vs. commercial tool
*alu*2	3.80	4.45	5.05	5.73	0.87	0.93	1.00	0.91	0.98	8.17/0.53 = 15.41
*alu*4	3.14	4.17	4.58	4.87	0.66	0.89	0.83	0.69	0.74	28.28/1.51 = 18.72
*apex*6	3.87	4.33	4.46	5.08	0.17	0.19	0.16	0.16	0.30	4.33/0.21 = 20.61
*apex*7	2.31	2.87	3.23	3.50	0.69	0.77	0.54	0.31	0.35	1.19/0.05 = 23.80
*C*1355	3.63	4.06	4.30	4.62	0.77	0.77	0.77	0.73	0.80	26.45/11.10 = 2.38
*C*1908	3.96	4.71	4.79	5.04	0.73	0.92	1.05	0.93	0.98	18.26/0.66 = 27.67
*C*2670	1.05	1.21	1.41	2.06	0.82	0.87	0.85	0.81	0.91	30.36/0.55 = 55.20
*C*499	3.79	3.90	4.02	4.09	0.70	0.75	0.81	0.76	0.83	26.52/11.02 = 2.40
*C*880	1.21	1.24	1.48	1.83	0.88	0.91	0.87	0.76	0.83	6.29/0.17 = 37.00
*frg*2	3.20	3.46	3.48	3.70	0.48	0.55	0.48	0.43	0.65	7.17/0.19 = 37.73
*i*5	0.05	0.10	0.25	0.32	0.44	0.50	0.50	0.50	0.74	1.32/0.03 = 44.00
*i*6	5.01	5.38	5.48	5.83	0.36	0.46	0.50	0.60	0.75	1.70/0.07 = 24.28
*i*7	5.26	5.30	5.36	5.37	0.83	0.88	0.92	1.37	1.45	3.15/0.07 = 45.00
rot	1.07	1.33	1.33	1.50	0.13	0.15	0.15	0.14	0.35	10.30/0.22 = 46.81
*term*1	2.33	2.88	2.98	4.36	0.66	1.50	2.16	1.80	1.82	1.13/0.10 = 11.30
too_large	0.73	1.07	1.26	1.91	0.73	0.75	0.81	0.81	0.87	4.53/0.36 = 12.58
vda	6.74	6.80	6.82	6.89	0.93	1.20	1.03	0.88	0.93	7.98/0.44 = 18.13
*x*1	0.50	0.52	0.74	1.03	0.53	1.00	1.06	1.33	1.36	1.67/0.11 = 15.18
*x*3	2.55	3.20	3.30	3.84	0.11	0.12	0.15	0.12	0.35	4.79/0.21 = 22.81
*x*4	3.73	4.24	4.40	4.60	2.5	3.00	3.16	2.83	2.85	1.95/0.07 = 27.86
AVG	3.01	3.52	3.73	4.07	0.52	0.59	0.59	0.55	0.77	195.54/27.67 = 7.07

Figure 4 shows the scatter plot comparing the SAT run times for the old method with both our techniques (described in Sections 4.1 and 4.2). The set of benchmark circuits used are a superset of those listed in Table III. Figure 4 shows that our methods perform consistently better, and are very effective for harder examples.

Table IV describes the effect of restricting the maximum number of literals in each new clause to a user-specified value k. For the results of Table IV,

we implemented both our techniques simultaneously. Column 1 lists the circuits used while column 2, 3, 4, and 5 list the clause overhead as a function of k. Columns 6, 7, 8, and 9 list the normalized runtime of our technique again as a function of k (compared to the old method). Column 10 lists the normalized runtime (with respect to the old method) when ACODCs are used. Finally, Column 11 reports the runtime of our method, the run-time for the commercial ATPG tool, and the ratio of the two. Since we compute an approximation of the CODCs in this technique, we choose $k > 5$ so that cubes of the ACODC are not removed. Further, since the number of clauses did not increase dramatically for $k > 5$ based on columns 2, 3, 4, and 5, this is a pragmatic choice. In general, the overheads for $k > 5$ are reasonable. Note that the ACODC method has a speedup of 23% on average for medium sized designs, compared to a 45% average speedup for the CODC method. This reduction is because ACODCs compute a subset of the CODCs, resulting in a reduced benefit. The reason for choosing ACODCs is that CODCs cannot be computed for large designs, while ACODCs can be computed [Saluja and Khatri 2004] much faster (25× faster on average) and with lower memory utilization (33× lower on average). This allows ACODCs to be used for large industrial designs, yielding a speedup of 31.5% for larger designs as we saw earlier. Note that our method is slower than the commercial ATPG tool for these small examples (by a factor of about 7×), since the SAT clause generation overhead dominates the total runtime. *However, as we saw earlier, for large designs, our method is 31% faster than the commercial ATPG tool on average.*

6. CONCLUSIONS

Boolean satisfiability (SAT) based formulations result in efficient techniques to solve the ATPG problem [Larrabee 1992; Stephan et al. 1996; Tafertshofer et al. 1997]. In these methods, we first transform the testability condition into an equivalent CNF formula. This formula is then solved using a SAT solver [Silva and Sakallah 1996; Moskewicz et al. 2001]. If the formula is satisfiable, the SAT solver returns a satisfying assignment, from which we can extract the test vector.

In this work we have presented two techniques to speed up SAT-based ATPG. In both techniques, we add clauses to the existing CNF formula in order to speed up the SAT solution process. In both techniques, these additional clauses are derived from the CODCs of the nodes of the circuit. We assume that CODCs of circuit nodes are computed before-hand, during technology independent logic optimization. As a result, they are available for the SAT-based ATPG tool, and there is no overhead in computing them. For large designs, for both techniques, we utilize Approximate CODCs (ACODCs) [Saluja and Khatri 2004], which can be computed efficiently for a design.

In our first technique we add clauses designed to speed up fault propagation. This is performed by augmenting the active clauses [Larrabee 1992] (which are written for nodes in the TFO of the node being tested) with don't care information.

In our second technique, we define new j-active variables and add new j-active clauses for selected nodes in the TFI of the fault node. These clauses are designed to speed up the fault justification process.

When using CODCs, we demonstrate an average improvement in runtimes of 41.5% when only our first technique was used. If both our techniques are used together, the average improvement in runtimes is 45%. When ACODCs are used, we demonstrate an improvement of about 31.5% in ATPG runtimes for large examples. When compared to a commercial ATPG tool, our method is 31% faster for large designs, but slower for small designs since the SAT clause generation overhead dominates the runtime for small designs. In the future, we plan to optimize our implementation, to further improve its performance. Further, we are looking at extending our technique to address sequential ATPG, by unfolding a sequential circuit in time, computing sequential Don't Cares and applying the same ideas outlined in the paper. In such a scenario, sequential don't cares will be used to additionally enhance the technique.

REFERENCES

BHATTACHARYA, D., AGRAWAL, P., AND AGRAWAL, V. 1995. Test generation for path delay faults using binary decision diagrams. *IEEE Trans. Comput. 44*, 3, 434–447.

BRAYTON, R. K., HACHTEL, G. D., MCMULLEN, C. T., AND SANGIOVANNI-VINCENTELLI, A. L. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers.

BRYANT, R. E. 1986. Graph based algorithms for Boolean function representation. *IEEE Trans. Comput. C-35*, 677–690.

CHANG, C. J. AND MAREK-SADOWSKA, M. 2002. ATPG-based logic synthesis: An overview. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 786–789.

CHANG, S. C. AND MAREK-SADOWSKA, M. 1994. Perturb and simplify: multi-level boolean network optimizer. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 2–5.

CHANG, S. C., VAN GINNEKEN, L. P. P. P., AND MAREK-SADOWSKA, M. 1996. Fast boolean optimization by rewiring. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 262–269.

FUJIWARA, H. AND SHIMONO, T. 1983. On the acceleration of test generation algorithms. *IEEE Trans. Comput. C-31*, 1137–1144.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Interactability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

GIZDARSKI, E. AND FUJIWARA, H. 2000. Spirit: satisfiability problem implementation for redundancy identification and test generation. In *Proceedings of the 9th Asian Test Symposium (ATS)*. 171–178.

GIZDARSKI, E. AND FUJIWARA, H. 2002. SPIRIT: a highly robust combinational test generation algorithm. In *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Syst.* 1446–1458.

GOEL, P. 1981. An implicit enumeration alogorithm to generate tests for combinational logic circuits. *IEEE Trans. Comput. C-31*, 215–222.

GUPTA, YANG, Z., AND ASHAR, P. 2001. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *Proceedings of the Design Automation Conference*. 536–541.

HUANG, C. Y., YANG, B., TSAI, H. C., AND CHENG, K. T. 2000. Static property checking using ATPG v.s. BDD techniques. In *Proceedings of the 2000 IEEE International Test Conference*. 309.

HUANG, S. Y., CHENG, K. T., AND CHEN, K. C. 2001. Verifying sequential equivalence using ATPG techniques. In *ACM Trans. Des. Autom. Electron. Syst*. 244–275.

KIRKLAND, T. AND MERCER, M. 1988. Algorithms for automatic test-pattern generation. *IEEE Design & Test of Computers*. Vol. 5. 43–55.

KUNZ, W. 1994. Multi-level logic optimization by implication analysis. In *IEEE/ACM International Conference on Computer-Aided Design*. 6–13.

KUNZ, W. AND PRADHAN, D. 1994. Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Syst. 13*, 9, 1143–1158.

LARRABEE, T. 1992. Test pattern generation using boolean satisfiability. *IEEE Trans. Comput.-Aid. Des. 11*, 4–15.

MCCLUSKEY, E. 1986. *Logic Design Principles: With Emphasis on Testable Semicustom Circuits*. Prentice-Hall.

MOSKEWICZ, M., MADIGAN, C., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*.

RADECKA, K. AND ZILIC, Z. 2001. Identifying redundant gate replacements in verification by error modeling. In *Proceedings of the International Test Conference*. 803–812.

RADECKA, K. AND ZILIC, Z. 2002. Identifying redundant wire replacements for synthesis and verification. In *Proceedings of the Asia and South Pacific Design Automation Conference and the International Conference on VLSI Design*. 517–523.

ROTH, J. 1966. Diagnosis of automata failures: A calculus and a method. *IBM J. Res. Develop. 10*, 278–291.

SAFARPOUR, S., VENERIS, A., DRECHSLER, R., AND LEE, J. 2004. Managing don't cares in Boolean satisfiability. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. Vol. 1. 260–265.

SALUJA, N. AND KHATRI, S. 2004. A robust algorithm for approximate compatible observability don't care (CODC) computation. In *Proceedings of the 41st Design Automation Conference*. San Diego, CA, 422–427.

SAVOJ, H. AND BRAYTON, R. 1990. The use of observability and external don't cares for the simplification of multi-level networks. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. Orlando.

SCHULZ, M. H., TRISCHLER, E., AND SARFERT, T. M. 1988. Socrates: A highly efficient automatic test pattern generation system. *IEEE Trans. Comput.-Aid. Des. 7*, 126–137.

SELLERS, F. F., HSIAO, M. Y., AND BEARNSON, L. W. 1968. Analysing errors with Boolean difference. *IEEE Trans. Comput. C-24*, 676–683.

SENTOVICH, E. M., SINGH, K. J., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1992. SIS: A system for sequential circuit synthesis. Tech. rep. UCB/ERL M92/41, Electronics Research Lab, University of California, Berkeley.

SHI, J., FEY, G., DRECHSLER, R., GLOWATZ, A., HAPKE, F., AND SCHLOFFEL, J. 2005. PASSAT: Efficient sat-based test pattern generation for industrial circuits. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. 212–217.

SILVA, M. AND SAKALLAH, J. 1996. GRASP-a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 220–7.

STEPHAN, P., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1996. Combinational test generation using satisfiability. *IEEE Trans. Comput.-Aid. Des. Integ. Circ. Systems 15*, 9, 1167–1176.

TAFERTSHOFER, P., GANZ, A., AND HENFTLING, M. 1997. A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 648–655.

TARJAN, R. 1974. Finding dominators in directed graphs. *SIAM J. Comput. 3*, 62–89.

VELEV, M. 2004. Encoding global unobservability for efficient translation to SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. 197–204.

ZHAOHUI, F., YINLEI, Y., AND MALIK, S. 2005. Considering circuit observability don't cares in CNF satisfiability. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. Vol. 2. 1108–1113.

Received June 2003; revised March 2007, June 2007; accepted July 2007