Efficient SAT-based Combinational ATPG using Multi-level Don't-Cares

Nikhil S. Saluja
Department of ECE
University of Colorado
Boulder, CO 80309
saluja@colorado.edu

Sunil P. Khatri
Department of EE
Texas A&M University
College Station, TX 77843
sunil@ee.tamu.edu

Abstract

In this paper, we present two combinational ATPG algorithms for combinational designs. These algorithms utilize the multi-level 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 run-times. When ACODCs are used, a speed-up of about 30% is obtained in the ATPG run-times for large designs. We compared our method against a commercial structural ATPG tool as well. Our method was slower for small designs, but for large designs, we obtained a 31% average speedup over the commercial tool.

1. Introduction

In order to ensure that a manufactured IC is defect 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.

Existing ATPG techniques for testing single stuck-at faults in combinational circuits fall under three classes. The first class consists of *structural* methods which perform a topological search of the circuit under test. In order to generate a test vector, these methods force a value discrepancy at the fault site (fault excitation). They then search for consistent values on circuit nodes such that the discrepancy is justified by a primary input assignment (fault justification). Finally they propagate the value discrepancy to some primary output of the circuit (fault propagation). Some of the well known structural methods are the D-algorithm [1], PODEM [2], FAN [3], TOPS [4] and SOCRATES [5]. The second class of techniques consists of *algebraic* methods which generate test patterns by manipulation of algebraic formulas. These methods produce a formula describing all possible tests for a particular fault and then simplify the resulting formula. The Boolean difference method [6] is an example of this class. The third class consists of *hybrid* methods. The Boolean satisfiability (SAT) method [7, 8, 9] is an example of this class. In this method, we start with a formula that encodes the testability condition (just as in the Boolean difference method), but instead of performing symbolic manipulation of this formula, we transform the formula into a conjunctive normal form (CNF) expression and run Boolean satisfiability (SAT) on the expression, to find a test.

In this paper we propose two techniques which significantly speed up SAT-based combinational 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. In our first technique we augment the active clauses of the nodes in the TFO of the fault site, to speed up the process of fault propagation. In the second technique we define new *j-active* variables for specific nodes in the TFI of the fault. Additional clauses using these j-active variables are then added to speed up the process of fault justification.

This paper could relatively easily be extended to address sequential ATPG, by simply unfolding a sequential circuit in time, 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 that were performed for this paper were conducted on combinational designs.

The rest of this paper 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 1. *The Boolean difference [10] of f with respect to x is defined as*

$$\frac{\partial f}{\partial x} = f_x \oplus f_{\overline{x}} \tag{1}$$

Definition 2. *Given a multi-level combinational Boolean network C, a single stuck-at fault f = f(x,$\mathcal{B}$) causes a node x in C to be permanently stuck at logic value $\mathcal{B}$ (where $\mathcal{B} \in \{0,1\}$). The faulty circuit, denoted by C_f is then C with the faulty node x assigned to $\mathcal{B}$.*

Definition 3. *The test for a node x stuck-at-0 is defined as*

$$(x) \cdot (\frac{\partial f}{\partial x})$$

where f is some primary output of the multi-level 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

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

Definition 4. *A conjunctive normal form (CNF) Boolean formula f on n Boolean variables $x_1, x_2, ..., x_n$ is a conjunction (logical AND) of m clauses $c_1, c_2, ..., c_m$. 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 5. *Boolean satisfiability (SAT) is the problem of determining whether a Boolean formula in conjunctive normal form (CNF) has a satisfying assignment.*

SAT is an NP complete problem [11]. Several heuristic approaches exist for efficient solution of SAT. Among these are Zchaff[12] and GRASP [13]. 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. In all cases, if a SAT solver determines that a formula is satisfiable, it also returns the corresponding satisfying assignment.

Definition 6. *The Observability Don't Care of node y_j in a multi-level Boolean network with respect to output z_k is*

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

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 [14]. This can also be denoted as

$$ODC_{jk} = (\overline{\frac{\partial z_k}{\partial y_j}}) \tag{2}$$

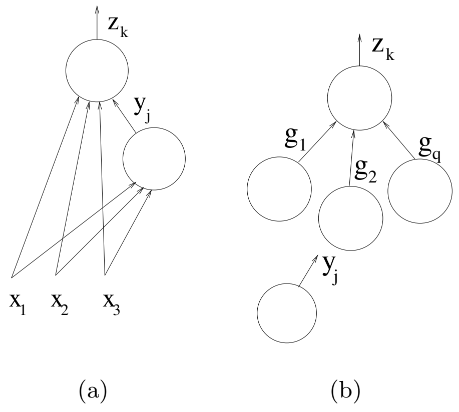

Figure 1: Don't Care example networks

In the network of Figure 1(a), z_k explicitly depends on y_j (i.e. y_j is a fanin of z_k) 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:

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

Once a node function is changed by minimizing [15] it against its ODCs, the ODCs of the other nodes must be recomputed. To avoid re-computation of ODCs during optimization, Compatible Observability Don't Cares (CODCs) [14] 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 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 [16].*

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.

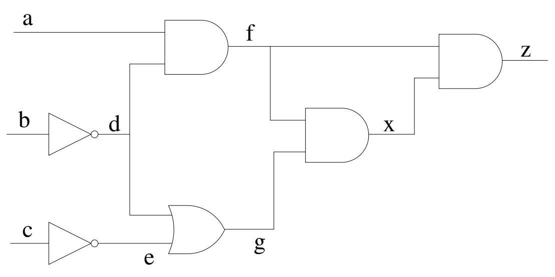

Figure 2: Dominators in Logic Network

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 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 primary output and the corresponding faulty circuit primary output.
- Clauses representing the logical OR of the clauses of the conditions 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* [7] are added to the composite CNF formula. These are described in Section 4.1.

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 [1] 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 [2] where the search space was restricted to the primary inputs of the circuit, resulting in a significantly more efficient ATPG technique. Techniques like FAN [3] 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 [4]. Structural ATPG techniques based on Boolean learning were introduced in [5, 17]. 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 the Boolean Difference Method [6]. 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 [7] and explored further in TEGUS [8]. 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.

EST [18] is an ATPG algorithm which utilizes structural search based on TOPS [4], augmented by Free BDDs [19] to keep track of previously encountered search states.

Another SAT-based technique was reported in [9]. 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 [9]. It would be interesting to see the performance of a method which combines our approach and that of [9].

In [20], the authors 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 [21, 22, 23] are similarly motivated. In [23], SAT is sped up for a circuit instance by labeling variables as *lazy* when they are determined to be non-controlling. 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 [21] uses don't care literals, which are treated differently during the solution process. A similar approach is reported in [22], 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 essentially orthogonal to those of [21, 22, 23], since the information used by these approaches does not utilize CODCs, but rather the structure of the circuit.

In [24, 25], the authors 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 [24, 25] lies in the fact that we utilize *approximate* CODCs (in order to handle large designs for which CODCs cannot be computed). Further, the application setting of our paper is widely different, resulting in the need to write very different clauses than [24, 25].

SPIRIT is a SAT-based ATPG tool [26] which implements structural concepts like unjustified lines and static learning. 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.

In [27], the authors 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.

4. Our Approach

Our approach utilizes a SAT-based formulation of ATPG. We utilize the efficient SAT solver Zchaff[12] to solve the SAT instance that arises from the ATPG problem. Our approach has two parts.

In the first part we augment the active clauses [7] in the TFO 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 [7], 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 TFO 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:

$$x_a \Rightarrow x \oplus x_f$$

Here x_f refers to the faulty value of the node x. 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 TFO 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 which can be added to the circuit are

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

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

In general, the clauses generated by the above expression may have a 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 TFO of the fault. Experimental results show an average improvement of 41.5% in the SAT run time due to the addition of these new clauses. The variable overhead is on average 0.1%, while the clause overhead is slightly more than 4%.

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 3 in Section 2) to:

$$\frac{\partial z_k}{\partial y_j} = (\frac{\partial z_k}{\partial g_1}) \cdot (\frac{\partial g_1}{\partial y_j}) \qquad (4)$$

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 4).

[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.

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 algorithm below. 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.

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

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

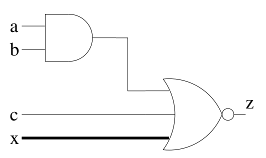

Figure 3: AOI-211 gate

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 [7] 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 [28] based computation. Therefore it is not possible to compute them for larger designs. Hence, for larger designs, we implemented a technique to compute approximate CODCs (ACODCs) [29], 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

We implemented both our techniques in SIS [30]. For our experiments we used the *mcnc91* and *itc99* benchmark circuits. Our experimental procedure consisted 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 replaced *full_simplify* with our ACODC [29] 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 run-time cost. Next we technology-mapped the circuit using the library *lib2.genlib* (using area as the cost function). Now for each uncollapsed fault in the design, we generated SAT clauses to test the fault, and then invoked Zchaff [12] to find a test. Our method was compared with SAT-based ATPG (without don't cares)[2] as well as a commercial ATPG tool. Comparisons with the old method were 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 were 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 used 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 was performed and *all* faults were tested using our deterministic procedure. This ensures that run-time comparisons are fair and objective. *The reason for this choice is that if random vector simulation was performed, the commercial tool and our tool may have tested a different set of faults during random vector simulation, making it impossible to draw objective conclusions from the results.*

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

[2]The original SIS ATPG algorithm uses a SAT solver internal to SIS. We changed this so that the SIS ATPG algorithm uses Zchaff [12] as the SAT solver. This method is referred to as the "old method" in this paper. It utilizes the notion of active clauses [7], but uses no don't care enhanced clauses like our method does.

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 were no aborted faults.* When ACODCs were used, these overheads reduce marginally. In all cases the clause and variable overheads are extremely reasonable.

Table 2 describes the ATPG results for large designs (for which CODCs cannot be computed). Therefore we used 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 run-time of our method, the run-time 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 run-time for *b22_C* is attributed to the presence of some hard faults, which our method tested but the commercial tool aborted on.

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 2. Figure 4 shows that our methods perform consistently better, and are very effective for harder examples.

Table 3 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 3, 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

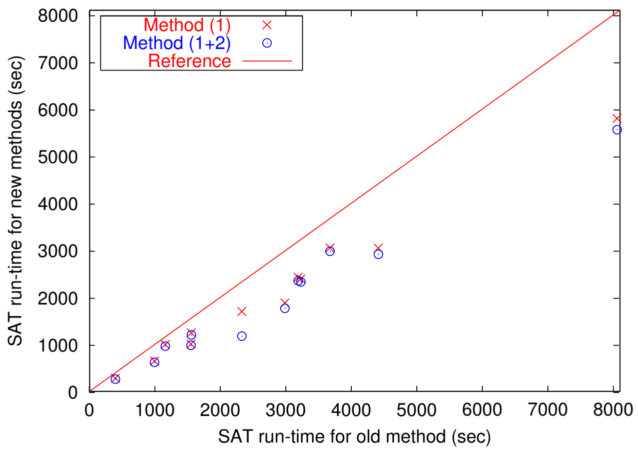

Figure 4: SAT run-time comparison using large examples

runtime (with respect to the old method) when ACODCs are used. Finally, Column 11 reports the run-time 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 chose $k > 5$ 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 was 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 [29] 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 [7, 8, 9]. In these methods, we first transform the testability condition into an equivalent CNF formula. This formula is then solved using a SAT solver [13, 12]. 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

circuit	flts	test	red	old clause	using CODCs				using ACODCs		
					cl % (1)	cl %(1+2)	old vars	var % (2)	cl % (1)	cl %(1+2)	var % (2)
alu2	976	968	8	1297k	5.12	5.73	411529	0.20	4.96	5.21	0.15
alu4	1833	1812	21	4191k	4.60	4.87	1366145	0.09	4.32	4.55	0.08
apex6	2153	2153	0	497k	4.46	5.08	202319	0.17	4.11	4.67	0.15
apex7	650	649	1	133k	3.23	3.50	57703	0.24	3.01	3.22	0.20
C1355	1337	1337	0	2643k	4.17	4.62	892533	0.03	3.83	3.99	0.02
C1908	1284	1282	2	2414k	4.84	5.04	857028	0.06	4.33	4.65	0.05
C2670	2304	2297	7	3145k	1.44	2.06	1260296	0.17	1.32	1.57	0.15
C499	1337	1337	0	2646k	3.90	4.09	981219	0.06	3.61	3.89	0.05
C880	1216	1216	0	824k	1.48	1.83	315654	0.11	1.22	1.45	0.10
frg2	2219	2213	6	780k	3.48	3.70	323958	0.06	3.18	3.35	0.05
i5	844	844	0	147k	0.23	0.32	70584	0.09	0.20	0.30	0.08
i6	1529	1529	0	195k	5.38	5.83	76996	0.15	4.56	4.92	0.12
i7	2096	2096	0	298k	5.26	5.37	121723	0.12	5.02	5.10	0.09
rot	1954	1953	1	1166k	1.34	1.50	479787	0.04	1.11	1.26	0.03
term1	498	494	4	131k	2.98	4.36	51436	0.32	2.77	3.43	0.22
too_large	810	803	7	518k	1.26	1.91	198527	0.20	0.99	1.34	0.15
vda	1092	1092	0	1127k	6.82	6.89	383695	0.02	5.96	6.01	0.02
x1	862	862	0	198k	0.52	1.03	85480	0.14	0.42	0.86	0.10
x3	2303	2303	0	508k	3.30	3.84	203281	0.15	3.03	3.26	0.11
x4	1126	1126	0	211k	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

Table 1: Clause and Variable Overheads for our Techniques - medium sized circuits

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

Table 2: Clause and Variable Overheads, and Runtime for our Techniques - large designs

circuit	clauses (1+2) w/ CODCs				norm. time (1+2) w/ CODCs				norm. time (1+2) ACODCs	our time (1+2) with CODCs ($k > 5$)
	k=3	k=4	k=5	$k > 5$	k=3	k=4	k=5	$k > 5$	k> 5	/commerc. time/ratio
alu2	3.80	4.45	5.05	5.73	0.87	0.93	1.00	0.91	0.98	8.17/0.53=15.41
alu4	3.14	4.17	4.58	4.87	0.66	0.89	0.83	0.69	0.74	28.28/1.51=18.72
apex6	3.87	4.33	4.46	5.08	0.17	0.19	0.16	0.16	0.30	4.33/0.21=20.61
apex7	2.31	2.87	3.23	3.50	0.69	0.77	0.54	0.31	0.35	1.19/0.05=23.80
C1355	3.63	4.06	4.30	4.62	0.77	0.77	0.77	0.73	0.80	26.45/11.10=2.38
C1908	3.96	4.71	4.79	5.04	0.73	0.92	1.05	0.93	0.98	18.26/0.66=27.67
C2670	1.05	1.21	1.41	2.06	0.82	0.87	0.85	0.81	0.91	30.36/0.55=55.20
C499	3.79	3.90	4.02	4.09	0.70	0.75	0.81	0.76	0.83	26.52/11.02=2.40
C880	1.21	1.24	1.48	1.83	0.88	0.91	0.87	0.76	0.83	6.29/0.17=37.00
frg2	3.20	3.46	3.48	3.70	0.48	0.55	0.48	0.43	0.65	7.17/0.19=37.73
i5	0.05	0.10	0.25	0.32	0.44	0.50	0.50	0.50	0.74	1.32/0.03=44.00
i6	5.01	5.38	5.48	5.83	0.36	0.46	0.50	0.60	0.75	1.70/0.07=24.28
i7	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
term1	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
x1	0.50	0.52	0.74	1.03	0.53	1.00	1.06	1.33	1.36	1.67/0.11=15.18
x3	2.55	3.20	3.30	3.84	0.11	0.12	0.15	0.12	0.35	4.79/0.21=22.81
x4	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

Table 3: Effect of Limiting Clause Sizes in our Techniques

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) [29], 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 [7] (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 have demonstrated an average improvement in run-times of 41.5% when only our first technique was used. If both our techniques are used together, the average improvement in run-times is 45%. When ACODCs are used, we have demonstrated an improvement of about 31.5% in ATPG run-times for large examples. When compared to a commercial ATPG tool, our method was 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.

In the future, we plan to extend our technique to address sequential ATPG, by unfolding a sequential circuit in time, 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

[1] J. Roth, *Diagnosis of automata failures: A calculus and a method*, vol. 10, pp. 278–291. IBM J. Res. Develop, 1966.

[2] P. Goel, "An implicit enumeration alogorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-31, pp. 215–222, 1981.

[3] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, vol. C-31, pp. 1137–1144, 1983.

[4] T. Kirkland and M. Mercer, "Algorithms for automatic test-pattern generation," in *IEEE Design & Test of Computers*, vol. 5, pp. 43–55, Jun 1988.

[5] M. H. Schulz, E. Trischler, and T. M. Sarfert, "Socrates: A highly efficient automatic test pattern generation system," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 126–137, 1988.

[6] F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson, "Analysing errors with boolean difference," *IEEE Trans. Computer*, vol. C-24, pp. 676–683, 1968.

[7] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 4–15, Jan 1992.

[8] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1167–1176, Sep 1996.

[9] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 648–655, Nov 1997.

[10] E. McCluskey, *Logic design principles : with emphasis on testable semicustom circuits*. Prentice-Hall, 1986.

[11] M. R. Garey and D. S. Johnson, *Computers and Interactability: A Guide to the theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the Design Automation Conference*, July 2001.

[13] M. Silva and J. Sakallah, "GRASP-a new search algorithm for satisfiability," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 220–7, November 1996.

[14] H. Savoj and R. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *27th ACM/IEEE Design Automation Conference*, (Orlando), June 1990.

[15] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[16] R. Tarjan, "Finding dominators in directed graphs," *SIAM Journal of Computing*, vol. 3, pp. 62–89, 1974.

[17] W. Kunz and D. 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*, vol. 13, pp. 1143–1158, Sep 1994.

[18] J. Giraldi and M. Bushnell, "EST: the new frontier in automatic test-pattern generation," in *Proceedings. 27th ACM/IEEE Design Automation Conference*, pp. 667–672, Jun 1990.

[19] S. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-26, pp. 509–516, June 1978.

[20] Gupta, Z. Yang, and P. Ashar, "Dynamic detection and removal of inactive clauses in SAT with application in image computation," in *Proceedings of the Design Automation Conference*, pp. 536–541, June 2001.

[21] F. Zhaohui, Y. Yinlei, and S. Malik, "Considering circuit observability don't cares in CNF satisfiability," in *Proceedings Design, Automation and Test in Europe (DATE) Conference*, vol. 2, pp. 1108–1113, Mar 2005.

[22] M. Velev, "Encoding global unobservability for efficient translation to SAT," in *Proceedings, International Conference on Theory and Applications of Satisfiability Testing*, pp. 197–204, May 2004.

[23] S. Safarpour, A. Veneris, R. Drechsler, and J. Lee, "Managing don't cares in Boolean satisfiability," in

Proceedings Design, Automation and Test in Europe (DATE) Conference, vol. 1, pp. 260–265, Feb 2004.

[24] K. Radecka and Z. Zilic, "Identifying redundant gate replacements in verification by error modeling," in *Proceedings. International Test Conference*, pp. 803 – 812, Oct-Nov 2001.

[25] K. Radecka and Z. Zilic, "Identifying redundant wire replacements for synthesis and verification," in *Proceedings, Asia and South Pacific Design Automation Conference and the International Conference on VLSI Design*, pp. 517–523, Jan 2002.

[26] E. Gizdarski and H. Fujiwara, "Spirit: satisfiability problem implementation for redundancy identification and test generation," in *Proceedings of the Ninth Asian Test Symposium (ATS)*, pp. 171 –178, Dec 2000.

[27] D. Bhattacharya, P. Agrawal, and V. Agrawal, "Test generation for path delay faults using binary decision diagrams," *IEEE Transactions on Computers*, vol. 44, pp. 434–447, Mar 1995.

[28] R. E. Bryant, "Graph based algorithms for Boolean function representation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–690, August 1986.

[29] N. Saluja and S. Khatri, "A robust algorithm for approximate compatible observability don't care (CODC) computation," in *Proceedings, 41st Design Automation Conference*, (San Diego, CA), pp. 422–427, June 2004.

[30] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.