CONCAT: CONflict Driven Learning in ATPG for Industrial designs

Surendra Bommu, Kameshwar Chandrasekar, Rahul Kundu and Sanjay Sengupta
{surendra.k.bommu, kameshwar.chandrasekar, rahul.kundu, sanjay.sengupta}@intel.com
Intel Corporation, Santa Clara, CA 95054

Abstract

In this paper, we propose a new search technique for ATPG, called CONCAT [1], which (a) is based on AND/OR reasoning, (b) integrates conflict driven learning, and (c) avoids over specification of test vectors. The technique works seamlessly (i) between Boolean and non-Boolean gates in industrial designs, (ii) across phases in latch-based designs, (iii) between justification and propagation tasks in sequential ATPG, and (iv) across faults in the fault list. Experimental results on combinational ISCAS circuits against SAT-based ATPG, show that we can reduce the test vector specification by upto 74%, with consistent improvement in performance and capacity. We integrated the CONCAT technique into Intel's existing ATPG tool, called Aztec, and obtained upto 67% speed-up and upto 14% more ATPG effectiveness on industrial designs.

1 Introduction

The relentless pursuit of Moore's law has led to a significant increase in the complexity of hardware designs. This poses serious challenges to the EDA community, in terms of improving the performance and capacity of the CAD tools. Specifically, the test generation complexity increases exponentially with the size of the circuit. Further, the cost associated with adding scan DFT for performance intensive applications is a prohibitive factor for performing full scan in industrial designs. This necessitates a robust sequential ATPG for partial scan industrial designs.

In this work, we focus on Automatic Test Pattern Generation (ATPG), which is a well studied problem in the Testing community. An ATPG tool generates test vectors to detect the faults in a design that model defects in the chip. These test vectors are used during manufacturing test to identify the defective chips. An improvement in the ATPG algorithm will potentially increase the fault coverage for the design and in turn improve the quality of the chips.

In ATPG: Given a circuit and a fault-list, we have to generate a test suite that either detects each fault or proves that it is undetectable. The single stuck-at fault model is widely used in the industry. We target each fault individually to generate a test vector. The other faults that are detected by the generated test vector are opportunistically dropped by fault simulation.

Various algorithms have been proposed, over the years, to tackle the basic ATPG problem. Roth's 5-valued D-algorithm [2], was the first complete algorithm introduced for ATPG. This was further augmented by using 9-valued algorithm [3] to address the inaccuracies encountered while handling sequential circuits. Later, Goel proposed PODEM [4], where the decisions are made only on the primary inputs in the circuit. This reduced the number of decision variables and lead to a faster ATPG. However, the technique suffers from over specification of the test vectors and late backtracking during local conflicts in the circuits. These issues were addressed in FAN [5] by making decisions on the headlines and using multiple back-tracing to choose a decision. In order to perform efficient justification during ATPG, Cheng [6] suggested the split circuit model, where the good machine and faulty machine values for all gates in the circuit are treated separately.

ATPG is generally guided by the testability measures to choose a decision during justification and propagation [7]. Several heuristics, such as distance based testability measures, probabilistic measures [8], SCOAP measures [9], super-gate based measures [10], correlation-based measures [11] were introduced as testability measures. These measures serve as heuristics and represent the relative difficulty of justifying a gate value to a control input or propagating a fault effect to an observe point. However, we have generally observed that there is no one measure that performs consistently better for all designs, if not for all faults in a design.

In [12], [13], the ATPG algorithm was accelerated by learning more necessary value assignments and unique sensitization points. They are used during ATPG to quickly identify a conflict or reach a solution for the search. Further, using Recursive learning [14], we could do an in-depth analysis to learn more implications. However, it is observed that we should cautiously avoid a rigorous application of these processing techniques to avoid run-time overhead and over-specification of vectors arising from unnecessary implied values.

On an orthogonal front, some explicit pruning methods were introduced to prune the search space during ATPG. In [15], equivalent search spaces were identified based on E-Frontiers and the corresponding search space was pruned or re-used during ATPG. Since we have to store the E-Frontier at each search point, this technique takes a hit in run-time and capacity when targeting huge designs. In [16], dependency directed backtracking and

1-4244-1128-9/07/$25.00 © 2007 IEEE

failure-driven assertion mechanisms were introduced to backtrack non-chronologically. In [17], the authors use non-chronological backtracking that can significantly reduce the number of backtracks in ATPG. Their technique is based on Boolean value splitting of each decision value and it is unclear if we can directly deal with multi-valued logic (including Zs) and propagation conflicts as necessary in ATPG. In [18], AND-OR reasoning graphs are used to identify necessary and non-conflicting assignments during ATPG. The learning is done specifically for each set of justification points to identify extra local implications. Similar reasoning is also done in Recursive Learning [14], to identify more implications during ATPG. Conflict driven heuristics and implication learning were also proposed in [19] for the D-algorithm which stores a local cache to learn the local implications. However, there is an inherent assumption that the circuit values are Boolean. Also, the techniques can learn only implications (single valued gates) locally. The limitations, we face, when applying these search-space pruning techniques to industrial designs are they: (i) intend to target Boolean valued logic (ii) increase the number of justification points (iii) do not scale well to sequential designs (iv) over specify the test vectors.

On a parallel front, Larrabee [20] introduced a SAT-based ATPG that models the circuit as a Conjunctive Normal Form (CNF) formula and uses SAT algorithms to solve the justification and propagation problems. The non-chronological backtracking and conflict-clause learning [21] fuelled a break-through in the Boolean Satisfiability area. With further engineering improvements to Boolean Constraint Propagation and conflict clauses learning [22], the performance and capacity of SAT solvers were significantly accelerated. The SAT-based techniques were borrowed in [23], [24] and [25] to improve ATPG. Other attempts have been made to incorporate the SAT-based techniques [26] [27] into the underlying algorithm to solve non-ATPG circuit problems. However, the potential challenges faced while using SAT for ATPG in industrial designs, are: (i) modeling complex gates, (ii) handling multi valued logic (iii) huge problem size in CNF (iv) over-specified solutions (v) handling propagation.

The main contributions of this work are:

1. Non-chronological backtracking based on AND/OR reasoning, in contrast to existing Boolean value splitting methods,
2. Conflict clause learning that is specific to the task of justification and propagation and,
3. Sharing conflict clauses that span across various phases among different faults.

We integrated the above conflict driven (CONCAT) techniques into Intel's ATPG tool (Aztec). The experiments were performed on combinational ISCAS circuits and compared with SAT-based ATPG. Then experiments were performed on industrial designs to compare CONCAT with Aztec and show the effectiveness and robustness of the approach.

In the next Section, we explain the underlying algorithm in Intel's ATPG tool: Aztec. In Section 3, we introduce the preliminaries necessary to understand the CONCAT techniques. In Section 4, we demonstrate non-chronological backtracking across AND/OR nodes. In Section 5, we illustrate conflict clause learning during justification and propagation. In Section 6, we demonstrate the effectiveness of the approach on industrial designs and conclude the paper in Section 7.

2 Background on Aztec: ATPG tool

In this section, we introduce the existing ATPG tool at Intel, Aztec, and explain the underlying algorithm with the help of an example. Aztec implements D-algorithm style ATPG in conjunction with single-path sensitization on a split-circuit model. It works directly on the circuit and handles complex gates such as XOR, BUS and memory elements. It uses multi-valued logic: [0, 1, U, Z], with weak and strong values around complex primitives. It is a sequential ATPG which can span across multiple time frames and is capable of handling partial scan designs. Static implication learning and computing testability measures are also built-in to the tool. Aztec supports a generic fault model which can handle stuck-at faults, bridge faults and transition faults.

The ATPG problem is partitioned into propagation and justification phases. First, the fault effect is propagated to an observable gate (across time frames, if required) through D-frontiers. During this propagation phase, the off-path J-Frontiers are collected. Then, the J-Frontiers are justified to find the test vector that detects the fault. When we reach a conflict, we backtrack and try alternative J-Frontiers or D-Frontiers to complete the search. Although, there are pros and cons for this technique theoretically, the ATPG was shown to perform well on large production designs and generates test vectors with minimal over-specification. Nevertheless, it should be noted that the proposed CONCAT techniques can be translated to any other ATPG algorithm as well.

We explain Aztec's algorithm with the help of the circuit segment shown in Figure 1. We use it as a running example throughout the paper. Although we use the split circuit model in the ATPG, we show the good and fault values at a gate together (example: D and D-bar to represent the fault effect) for the ease of explanation.

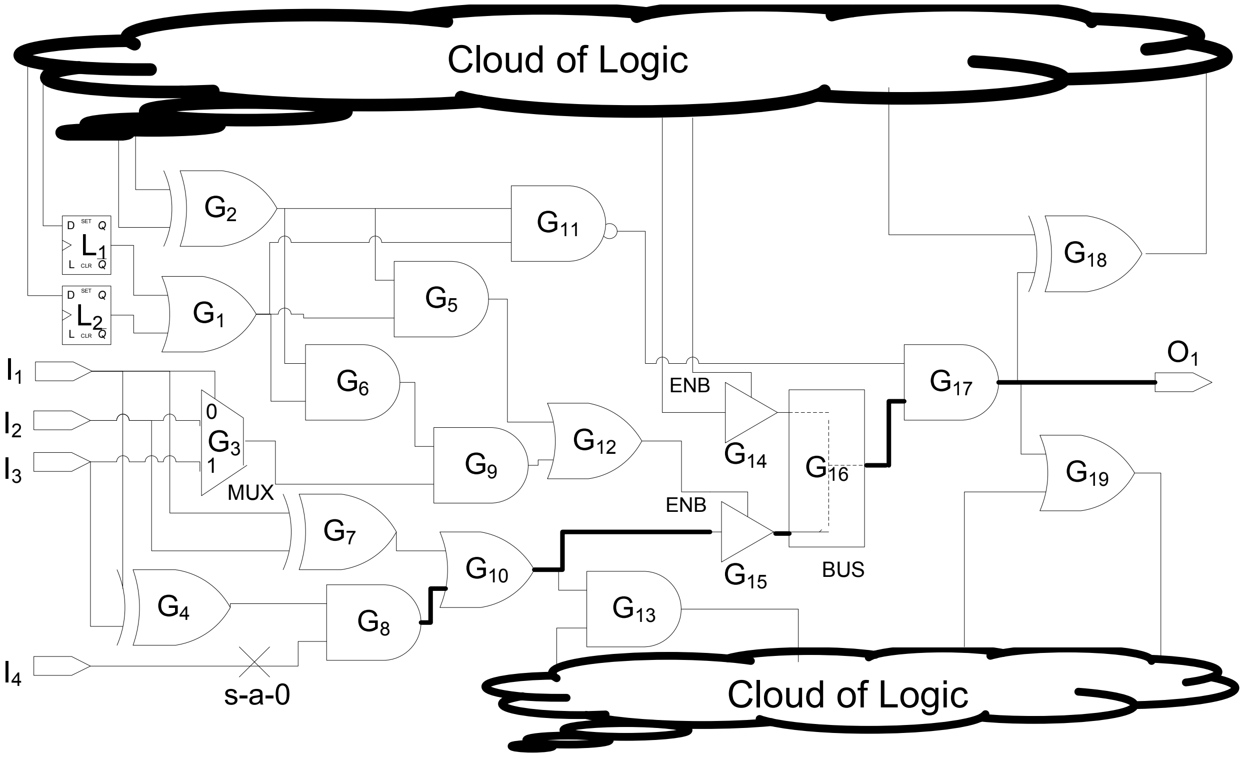


Figure 1: Running Example to demonstrate CONCAT

Consider the fault I_4 stuck at 0. First, we try to propagate the fault effect to the observe point O_1. During this propagation phase, the D-frontiers are identified through the sensitized path, G_8-G_{10}-G_{15}-G_{16}-G_{17}-O_1. The alternative D-Frontiers G_{13} (to G_{15}) and G_{18}, G_{19} (to O_1) are added to a stack for later manipulation. The corresponding J-frontiers to justify the off-path inputs, G_4=1, G_7=0, G_{12}=1, G_{14}=Z, G_{11}=1 are added to a J-Frontier list. They are mandatory to sensitize the propagation path. Then, in the justification phase, we justify the J-frontiers one-by-one. Let us say we first pick the J-frontier G_{12}=1 to be justified. We can justify G_{12}=1 with G_9=1 or G_5=1. If we choose G_9, it results in implications G_6=1, G_1=1, G_2=1, G_3=1. The new J-frontier list is: G_4=1, G_7=0, G_1=1, G_2=1, G_3=1, G_{14}=Z, G_{11}=1. In this way, we try to justify the gate values all the way to primary inputs or control points.

3 Preliminaries

In this section, we demonstrate the AND/OR reasoning framework, Implication Graph construction and conflict diagnosis techniques that facilitate CONCAT techniques.

3.1 AND/OR Reasoning Framework

We demonstrate the AND/OR relationship that manifests while propagating the fault effect and justifying a gate value during ATPG. In the propagation phase, when we propagate the fault effect through a D-frontier (say parent), the fanouts of the D-frontier (say children) form the next potential candidates to propagate the fault. Since we have to choose any one of the children D-frontiers to propagate the fault effect, they share an OR-relationship. The parent D-frontier and the chosen child D-frontier are linked by an AND-relationship, since the fault effect has to propagate through both the gates to reach an observable point. The fault effect sensitization path can be seen as a set of D-frontier nodes linked by an AND relationship. Also, the off-path J-Frontiers that need to be justified to propagate the fault across a D-frontier has an AND relationship with that D-Frontier.

In the justification phase, we will see AND/OR choices as demonstrated in the earlier examples in Figure 1: We can justify G_{12}=1 by choosing either G_9=1 OR G_5=1. Next, consider the justification of G_9=1 in the circuit. It implies G_6=1, G_1=1 G_2=1 and G_3=1. Now, we have to justify G_1=1 AND G_2=1 AND G_3=1.

3.2 AND/OR Implication Graph

The AND/OR reasoning framework for the example in Figure 1 will be clear from the graph in Figure 2. The nodes in the box represent the propagation phase and the nodes outside the box represent the justification phase. For the ease of explanation, we use the following conventions in the graph: (i) the rectangles represent D-frontier nodes, hexagons represent J-frontier nodes and ovals represent other implication nodes. (ii) we use solid lines to show AND relations, dotted lines to show OR relations and arrows to show implications in the graph (iii) a node with thicker boundary represents the chosen decision among other OR choices. Each decision is characterized by a decision level, which represents the chronological order of decision making. In Figure 2, nodes N_1, N_5, N_{11} were chosen decisions made in the order 1, 2, 3 (prefixed by @). Each implication node gets the maximum decision level among all its fanins. We refer a node and the gate value it holds interchange-ably when the reference is clear from the context.

For the fault in Figure 1, the propagation path is represented by the nodes N_1-N_3-N_5-N_7-N_9-N_{11} in the propagation phase. The D-frontier choices, for example G_{13} at node N_5, are linked by dotted lines. The J-Frontiers resulting from the propagation phase are linked by solid lines to show the AND-relationship between the D-Frontier and off-path J-Frontier: Ex. (G_{12}=1) to N_5.

In justification phase, consider justification of G_{12}=1 at node N_6. The OR choices G_9=1 OR G_5=1 are linked by dotted lines. We choose G_9=1 in the example (thick boundary line) and keep G_5=1 as an alternative. Next, consider the justification of G_9=1 at node N_{12}. The resulting implications, G_3=1, G_6=1, G_1=1 and G_2=1, are linked by an AND relationship as shown by solid lines. We also maintain implication relations: [(G_6=1)$\rightarrow$(G_1=1)] and [(G_6=1)$\rightarrow$(G_2=1)] in the graph as shown by arrows from the implicant to the implication.

Continuing from previous section, we have J-Frontiers: G_4=1, G_7=0, G_1=1, G_2=1, G_3=1, G_{14}=Z, G_{11}=1. As shown in Figure 2, we choose to justify G_1=1 at N_{14} and make the decision L_1=1 with an OR choice L_2=1 at decision level 5. Then, we try to justify the J-Frontier G_3=1 at N_{16} and make the decision (I_1=0, I_2=1) with an OR choice (I_1=1, I_3=1) at decision level 6. In our implementation, we effectively create the graph nodes, only for the assigned gate values in the circuit. We create nodes specific to a time frame in order to handle the different time frames in sequential ATPG. The active nodes are shown by node numbers in the Figure 2 in their chronological order of occurrence. The OR choices do not have an explicit implication graph node creation and are maintained in a separate stack. Since the implication graph is only constructed for the portion of the circuit that is active during ATPG, we see that the size of the implication graph is generally small as compared to the size of the circuit.

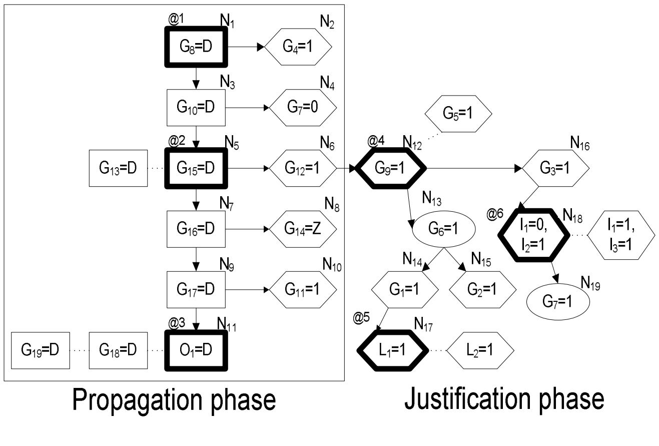


Figure 2: AND/OR Framework for ATPG

3.3 Conflict-diagnosis

The main intention of using an implication graph is to identify the set of *implicants* that imply a specific value on a gate in the circuit. For each gate value in the circuit, there is a corresponding node in the implication graph. Starting from a given node, if we traverse backward in the implication graph, we can identify the set of implicants that imply the gate value in the node.

When we reach a conflict scenario in ATPG, starting from the conflicting nodes, we can traverse backwards in the implication graph to identify the set of implicants that are actually responsible for the conflict. This step is called the *conflict diagnosis* step. Figure 2 actually shows a conflict scenario. The gate values in nodes N_{19} and N_4 conflict for gate value at G_7 and represent the conflicting nodes. During backward traversal for conflict diagnosis, we start from the conflicting nodes and stop at a node if it is (i) a decision node or, (ii) an implication node at a lower decision level than the highest decision level of the conflicting nodes. In the above example, we identify that N_{18} and N_4 are responsible for the conflict. The node N_4 is the actual *reason* for the conflict arising from the decision made at N_{18}. We refer the reader to [28] for more details on conflict diagnosis and other ways of collecting the reasons for a conflict.

4 Non-chronological backtracking

The basic idea is as follows: Let J be a J-frontier and R be the set of implicants for J. Let $C_1, C_2, \ldots C_k$ be the OR choices that can justify J. If each of C_i leads to a conflict and R_i is the corresponding reason set for the conflict at C_i, we can backtrack to decision made @ l where, l is the maximum decision level among all the decisions levels of nodes in sets $\{R, R_1, \ldots R_k\}$.

Consider the scenario in Figure 2, where we are currently at decision level 6, justifying G_3=1 (at node N_{16}) i.e. R= {N_{16} @4}. We have two choices [(I_1=0, I_2=1), (I_1=1, I_3=1)] to justify G_3=1. The current decision is C_1: (I_1=0, I_2=1). It is obvious that N_{19} and N_4 represent conflict values at gate G_7 and we have to backtrack. After we do a conflict diagnosis step, as explained earlier, we identify that N_{18}: (I_1=0, I_2=1) and N_4: (G_7=0) are the implicants for the conflict. Therefore, (I_1=0, I_2=1) cannot be justified because of the reason-set: R_1={N_4 @1}.

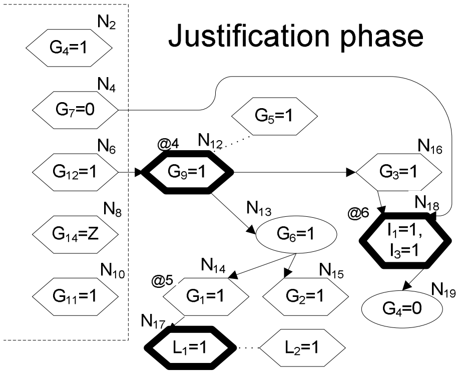


Figure 3: Second choice to justify G3=1

Now, we make the next choice, C_2: (I_1=1, I_3=1) as shown in Figure 3 (only the required and changed nodes in the justification phase are shown). This again leads to a conflict at gate G_4. Now, R_2 = {N_2 @1} is the reason for this conflict. Among all the reasons identified so far {N_{16} @4, N_4 @1, N_2 @1} the maximum decision level is 4 and the corresponding decision node is N_{12}. So we can backtrack to N_{12} @4 directly and make the other alternative choice G_5=1 at N_{12}. We are effectively skipping the OR choice L_2 at decision level 5 and backtracking non-chronologically from decision level 6 to decision level 4. In the absence of the conflict diagnosis step, the ATPG will backtrack to decision level 5 and unnecessarily try to justify G_1=1.

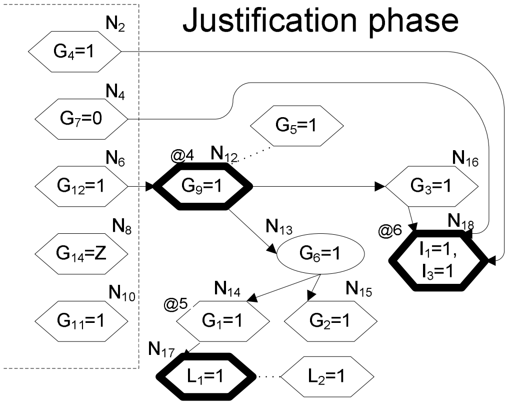


Figure 4: Non-chronological backtracking

In our implementation, we re-use the justification decision graph node to make all choices and collect each of the *conflicting reasons at its fanins*. After all choices are exercised, we perform conflict diagnosis on the common justification graph node to backtrack. For the example in Figure 3, the reason set R={N_{16}} is a fanin to N_{18} initially. After the first decision (I_1=0, I_2=1) leads to a conflict, we add the reason set R_1={N_4} as fanin to N_{18}. After the second decision (I_1=1, I_3=1) leads to a conflict, we add the reason set R_2={N_2} as fanin to N_{18} (see Figure 4). Since all choices have been exercised at N_{18}, we do a conflict diagnosis on N_{18}. The conflict implicants are {N_{12}@4, N_2@1, N_4@1} and we backtrack to the most recent decision node, which is N_{12} @4.

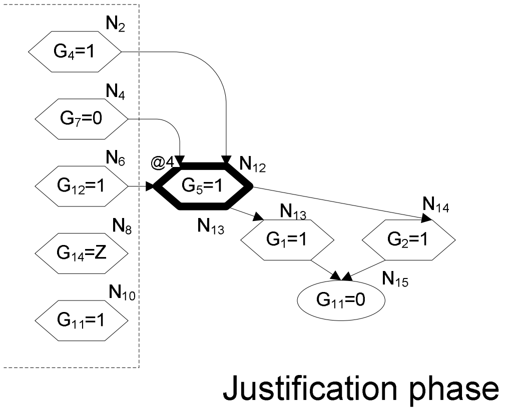


Figure 5: Second Choice to justify G12=1

Let us continue with ATPG to look at another scenario where we backtrack from the justification phase to propagation phase. In Figure 4, we have backtracked to node N_{12} with OR choice G_5=1. At this point, we are actually justifying G_{12}=1 with choices G_9=1 and G_5=1. The J-frontier reason-set is R={N_6}, which is a fanin to N_{12}. In the earlier scenario, we saw that the decision G_9=1 lead to conflict and the corresponding reason set is R_1 = {N_4, N_2}. When we backtrack to N_{12}, these nodes are added as fanins to N_{12} (see Figure 5).

In Figure 5, we exercise the next choice: (G_5=1) at N_{12}. We imply the value, G_5=1 in the circuit and add the corresponding implication nodes to the graph (please refer Figure 1). It can be seen that nodes N_{15} and N_{10} conflict with each other for gate value at G_{11}. After conflict diagnosis on N_{15} and N_{10}, we identify that R_2={N_{10}} is the reason set for choice (G_5=1) to fail. We add the node N_{10} as a fanin to N_{12} (see Figure 6). After doing a conflict diagnosis on node N_{12}, it is identified that we can backtrack to N_5 @2. We have pruned the search space for the D-frontier choices at node N_{11} @3.

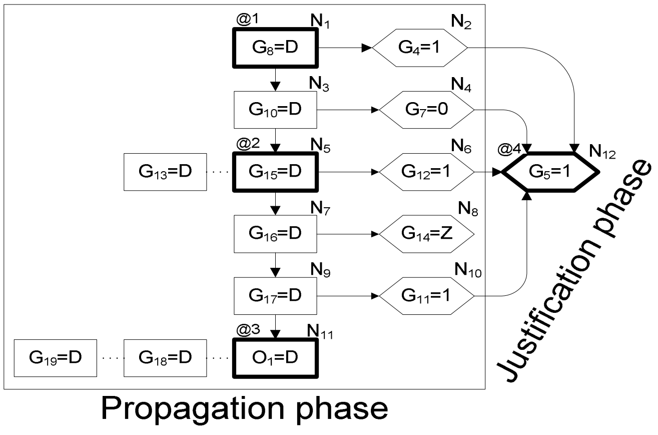


Figure 6: Non-chronological backtracking

Similar reasoning holds for D-frontiers within the propagation phase as well. However, due to the nature of ATPG algorithm (propagation first), we will *effectively* perform chronological backtracking. When a particular D-frontier choice leads to a conflict, we will backtrack to the next available choice at its sibling or parent, irrespective of doing conflict diagnosis or not.

It should be noted that our search is based on different ways of solving an objective and we revert the conflicting choices back to X. This makes it independent of the value system. In Figure 6, we backtrack because choices (G_9=1) and (G_5=1) to justify (G_{12}=1) fail. In Boolean value splitting [16] [17] [20] [21], we should exercise (G_9=0) after (G_9=1) and perform non-chronological backtracking if both fail. The earlier techniques do not address: (a) multi-gate decision (Figure 3) (b) non-Boolean decision values and (c) non-chronological backtracking during propagation. They may lead to additional justification points (G_9=0) and hence, generate over-specified test vectors.

5 ATPG Conflict Clauses

In addition to non-chronological backtracking, we learn from conflicts during justification and propagation.

5.1 Justification conflict clauses

Let us again consider the J-Frontier *J* with reason set *R*:

$$R \Rightarrow J$$

Let $C_1, C_2, \ldots C_k$ be its OR (V) choices:

$$J \Rightarrow \left(\overset{k}{\underset{i=1}{\vee}} C_i \right)$$

If all choices lead to a conflict, and R_i is the reason set for each C_i leading to a conflict:

$$C_i \Rightarrow \neg R_i$$

By substitution and transitive property:

$$R \Rightarrow \left(\overset{k}{\underset{i=1}{\vee}} \neg R_i \right)$$

Therefore, due to the implication property, it is concluded that $R \wedge \left(\overset{k}{\underset{i=1}{\wedge}} R_i \right)$ represents a conflict space and represents the *justification conflict clause*. Essentially, the nodes collected by the conflict diagnosis step at the J-frontier (example cases in Section 3) represent this conflict space. When we encountered a conflict at J-frontier G_3=1, nodes N_{12}, N_4, N_2 were identified as the implicants during the conflict diagnosis step (see Figure 4). In that case, R={G_9=1}, R_1={G_7=0}, R_2={G_4=1}, the corresponding conflict space is {G_9=1, G_7=0, G_4=1} which is non-trivial learning. The gate values from the implicant nodes identified from the conflict diagnosis step at the J-Frontier are stored in a conflict database as a *conflict term*. When we encounter the same combination of gate-value pairs again during ATPG, we can backtrack right away. Storing the gate value pairs as a term (not complementing to analogous SAT clauses), helps us to handle the multi-valued system elegantly.

We implemented the two literal watching scheme, similar to zchaff [22]. Since we maintain a conflict term, the watched literals are two unsatisfied literals in ATPG. We perform unit-implication on literals only where a complement exists. We *explicitly* avoid the justification of the unit implications, arising from the conflict clauses, to avoid over specification of test vectors. Any conflicts arising from implications will be flagged to the ATPG.

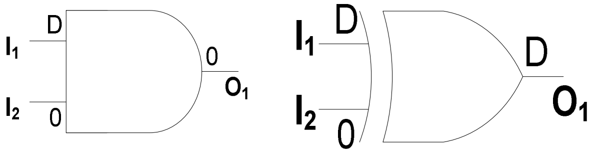
5.2 Propagation conflict clauses

Consider a D-Frontier D_p (parent) and its OR choices (children) $C_1, C_2, \ldots C_k$. For the sake of discussion, let us consider that the children are gates with a fault effect at its input, after implying values at D-Frontier.

We can classify the D-Frontier children into three types, based on the following criteria:

1. Propagation blocked: These are gates which block the fault effect from D_p. Consider the AND gate in Figure 7 (A). We are not able to propagate the fault effect because of the off-path of the AND gate being 0 i.e. the reason for the fault effect being blocked is $I_2=0$.

2. Propagation constrained: These are cases where the fault effect can be propagated as a D or D-bar and one of them is blocked. For example, in Figure 7 (B), the output of the XOR gate can take values D or D-bar. But, the D-bar value is constrained because of $I_2=0$.



(A) Propagation Blocked (B) Propagation Constrained

Figure 7: Propagation Constraining scenarios

3. Off-path unjustifiable: These are the children which become new D-frontiers during propagation. However, we are not able to justify the off path input values during the justification phase. The reasons for not being able to justify the off-path J-frontier values can be obtained by the conflict-diagnosis step in justification. These reasons are the propagation constraining nodes for the D-frontier. In the previous section, we saw that we are not able to justify the off-path input of the D-frontier, G_{15}, i.e. G_{12}=1. The reasons obtained during the J-frontier conflict diagnosis step, i.e. N_4, N_2 are the propagation constraining nodes for the D-frontier, $G_{15.}$

When we backtrack to the parent D-frontier (D_p), we collect the reasons for not being able to propagate the fault effect through all its children, say $R_1, R_2, ... R_k$. It is concluded that if we choose the D-Frontier D_p again and $R_1, R_2, ... R_k$ are true, then we will not be able to propagate the fault effect to an observe point from D_p. Therefore, $R_1, R_2, ... R_k$ represent a conflict clause when we choose D_p. We store the conflict clause $\{D_p, R_1, R_2, ... R_k\}$ as a *propagation conflict clause*.

In the case of a propagation conflict clause, we do not perform any unit implications. We just check if a conflict space has been reached or not. When D_p is chosen, if $R_1, R_2, ... R_k$ become true, then we signal a conflict to the ATPG engine. We store $\{D_p, R_1, R_2, ... R_k\}$ as a term and use the 2-literal watching scheme in a slightly different way to handle the propagation conflict clauses. The D-frontier D_p is always made the first watching literal and serves as an index to trigger the corresponding conflict clauses. The second watching literal is chosen from one of the remaining unsatisfied literals. When a D-frontier is chosen, the first watched literal automatically indexes to the set of the reasons that will make that D-frontier fail. The second watched literal tries to find an unsatisfied literal. If none is available, we signal a conflict to ATPG.

5.3 Conflict Clause sharing

Although we learn the conflict clauses for each fault, it is possible to share the learning across other faults as well. If a clause is learnt from a conflict diagnosis step, where all the gate values involved are good values, then the learning is independent of the fault. We call those clauses as *fault independent clauses*. On the other hand, if any of the gate values involved in the conflict diagnosis step are faulty values, then we call the clause a *fault dependent clause*. We can retain the fault independent clauses for all the other faults. In our implementation, we have a fault independent clause data base and a fault dependent clause data base. The fault dependent clause data base is emptied after each fault is processed, whereas, the fault independent clause database is shared across faults.

5.4 ATPG restarts

Since the ATPG tool is sensitive to the testability measure heuristics, a fault may be hard-to-detect for a particular heuristic and trivially detected by a different heuristic. We restart the ATPG, after searching for a fixed number of backtracks, with different testability measures to target these test cases. In the case of ATPG restarts, the fault dependent clauses are also shared across different restart runs of ATPG for the same fault. The conflict clauses also guarantee that the same search space is not re-visited during restart ATPG runs and hence make the restarts more efficient.

6 Experimental Results

We integrated the proposed conflict learning (CONCAT) techniques into the basic Aztec ATPG tool. We refer to CONCAT with Aztec as CONCAT and the basic Aztec as Aztec in this section. We perform three sets of experiments: (i) CONCAT Vs SAT based ATPG on ISCAS combinational circuits (ii) CONCAT Vs basic Aztec on large industrial designs with high fault coverage and, (iii) CONCAT Vs Aztec on high combinational depth industrial designs with hard-to-detect faults.

6.1 CONCAT Vs SAT on ISCAS

We use Intel's state-of-the art SAT solver, which is also capable of generating clauses for all gates in a given circuit. It uses 3 variables to encode each gate value. Even X values are encoded into the variables. For SAT-based ATPG, we create a Boolean difference circuit for the good and faulty machines to represent the ATPG problem and generate the clauses. The faulty machine clauses are created only for the gates in the fanout cone of the fault. The CNF is fed to the SAT solver and we try to justify a 1 at the output of the Boolean difference circuit. If the SAT solver returns a solution, then the fault is testable and the good value inputs represent the required test vector. If the SAT solver reports unsatisfiable, then the fault is untestable. We compare the results with CONCAT. Similar to SAT, we also enabled random restarts in CONCAT with different testability heuristics. We performed the experiments on combinational ISCAS circuits and the results are tabulated in Table 1. Both SAT based ATPG and CONCAT were run without fault dropping in the same environment. The experiments were run until all the faults were resolved, except in c6288.

In Table 1, the first column specifies the name of the circuit. The second and third columns tabulate the time taken by SAT and CONCAT. Columns 4 and 5 represent the memory required to run CONCAT and SAT on a common environment. Columns, 6 and 7, show the % of specified bits among all the test vectors generated by

Circuit	Time (s)		Mem (MB)		Spc bits (%)		Δ
	SAT	CON	SAT	CON	SAT	CON	(%)
C432	1	<1	50	32	55	30	45
C499	3	2	50	33	91	78	14
C880	2	<1	50	33	33	15	54
C1355	13	1	52	33	97	87	10
C1908	18	1	53	34	88	43	50
C2670	18	1	56	36	23	13	43
C3540	217	2	57	36	46	28	38
C5315	32	2	59	38	53	13	74
C6288	2280*	903	66	48	84	80	5
C7552	278	7	79	45	25	9	62

*28 aborts **Table 1: SAT Vs Concat on ISCAS**

SAT and CONCAT. The last column shows the decrease in the % specified bits achieved by CONCAT. A lesser % specified bits is beneficial for test vector compaction and hardware compression purposes.

It is seen that CONCAT consistently performs better than SAT on all the test cases. The highlight is c6288 were CONCAT finishes for all faults in 903 seconds, but SAT is left with 28 aborts, even after 2280 seconds. It is also seen that SAT based ATPG requires more memory than CONCAT to perform ATPG. From the last column, it is seen that CONCAT is able to obtain a better test suite than SAT. As mentioned earlier, the nature of SAT solving leads to specifying more variables and hence over specified test vectors. We are able to achieve upto 74% reduction in the number of specified bits on the ISCAS combinational circuits as seen in c5315.

6.2 CONCAT Vs Aztec on Industrial designs

To quantify the overhead of CONCAT techniques, non-chronological backtracking and conflict clause learning, we compared CONCAT with Aztec for designs with high fault coverage. We ran experiments on high to moderate scan industrial designs with upto 1.3 million gates with fault dropping after each vector is generated.

The results are tabulated in Table 2. The first column represents the design. The next column shows the number of gates in the circuit. Columns, 3 and 4, show the fault coverage numbers, which is the number of detected faults over the total number of testable faults. Columns, 5 and 6, report the ATPG effectiveness numbers which represents the number of faults resolved (including untestable faults) over the total number of faults. Columns, 7 and 8 show the #patterns generated by the run for Aztec's fault coverage. Columns 9 and 10, show the memory foot print, Columns, 10 and 11, report the time and the last column signifies the improvement in run-time achieved by CONCAT over the Aztec.

From Table 2, it is seen that CONCAT is able to achieve slightly higher fault coverage than Aztec, and the overheads of the new technique on the tool are minimal. In particular, we see that the patterns generated by CONCAT are similar in length or better for most test cases. This validates the assertion that the new techniques preserve the minimal over-specification strength of D-algorithm. The increased effectiveness (even on high coverage test cases) shows the improvement in robustness of the algorithm with the new techniques. For similar fault coverage, upto 33% improvement in run-time is obtained due to CONCAT techniques as seen for D2 in Table 2. On the other hand, for D1 and D5, CONCAT achieves about 1% more ATPG effectiveness than Aztec. This means we are able to identify more untestable faults than Aztec. It is also seen that CONCAT is able to achieve the fault coverage of Aztec with a lesser # patterns in most of the designs. Since the fault-dropping changes the order of faults targeted by Aztec and CONCAT, it has a slightly adverse effect in D1 pattern count for CONCAT. The trade-off is in terms of the memory foot print that reflects the maintenance of an additional implication graph to do the AND/OR reasoning. Nevertheless, the increase in the size of the implication graph is contained and is within admissible limits for the tool. This increase in memory also correlates with the improvement in performance for each of the runs.

We demonstrate the incremental effect of each of the proposed techniques: non-chronological backtracking (Concat 1), justification conflict clauses (Concat 2), propagation conflict clauses (Concat 3), and random ATPG restarts (Concat 4) in Figure 8 with experiments on one of the latest CPU products. The curve for Aztec remains flat showing that we are not able to achieve any increased effectiveness, even though we increase the number of backtracks. The Concat ATPG performs better than Aztec with improved effectiveness and faster performance. Although, we see the highest effectiveness

Design	#Gates	Fault Cov (FC) (%)		Atpg Eff (%)		#patterns for Aztec's FC		Memory (MB)		Time (s)		%ΔTime
		Aztec	Concat	Aztec	Concat	Aztec	Concat	Aztec	Concat	Aztec	Concat	
D1	193K	95.50	95.64	97.52	98.56	583	601	212.04	240.14	1265	1068	15.57
D2	1M	82.54	82.76	97.32	97.62	140	124	791.02	976.20	1798	1196	33.31
D3	504K	91.10	91.13	97.43	97.73	481	343	416.21	466.32	1308	1184	9.48
D4	1.3M	66.89	66.87	95.95	95.84	154	139	848.64	998.72	476	372	21.84
D5	422K	93.73	93.85	98.44	99.44	674	641	359.25	419.73	938	859	8.42

Table 2: Aztec Vs Concat

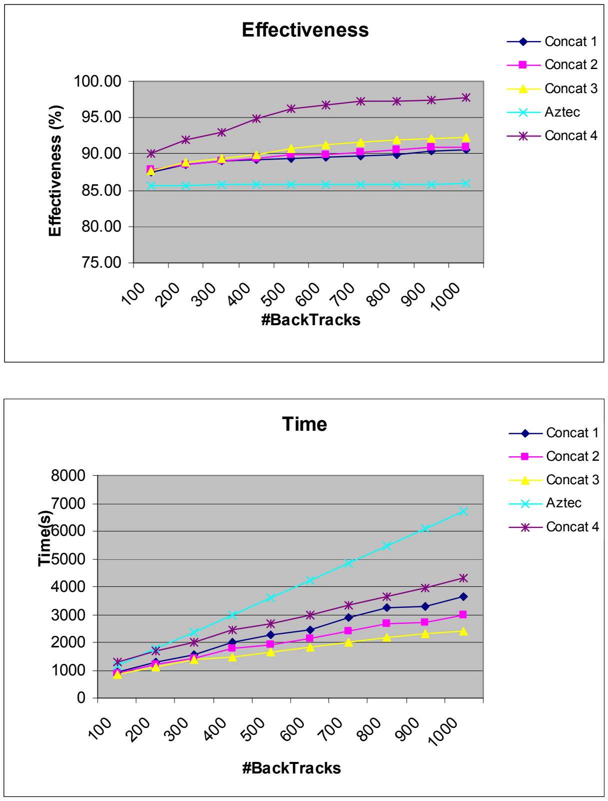


Figure 8: Concat Vs Aztec

gain in Concat 4, it slows down the ATPG. It is generally recommended that we use the lower Concat levels to screen out the easy to detect faults and then target the remaining hard to detect faults using Concat 4.

6.3 Concat Vs Aztec on hard-to-detect faults

We conducted another set of experiments, where we target the functional blocks in recent microprocessor designs with high combinational depth, which have hard-to-detect faults. The results are tabulated in Table 3, with columns labeled similar to those in Table 2. For each of these designs, we see a consistent improvement in fault coverage and an even higher improvement in the ATPG effectiveness. For example, in SD4, we see about 3 % improvement in fault coverage and 6 % improvement in ATPG effectiveness. This shows that we are able to improve the quality of the test suite, by detecting more testable faults and also, prove that more faults are untestables. It should be noted that, we are also able to achieve a 54% decrease in run-time for this test case.

On the other hand, in SD6, CONCAT achieves around 11% improvement in fault coverage. CONCAT is generating more test vectors to improve the quality of the test suite. The additional time required to search for more test vectors translates to 9% more run time as seen in the last column of Table 3. It is generally seen that CONCAT performs significantly better than Aztec in designs with high combinational depth. These results show that the AND/OR reasoning helps to increase the fault coverage of the test suite and improve the performance of the ATPG tool.

7 Conclusions & Future work

We proposed a new search technique with conflict driven learning that is suitable for an industrial sequential ATPG tool on production designs. The learning is based on AND/OR reasoning of the propagation and justification tasks and hence can handle non-Boolean logic in complex gates directly. We are able to perform non-chronological backtracking based on the reasoning framework, which prunes the search space during ATPG. We learn justification and propagation conflict clauses that help to avoid re-visiting the same conflict space multiple times. We show that it is possible to learn certain fault independent conflict clauses that can be shared across multiple faults. Experimental results show that we are able to reduce over-specification as compared to SAT-based ATPG and increase the ATPG effectiveness over the existing ATPG tool at Intel.

As a by-product of the reasoning framework, we can root-cause the untestability of an untestable fault. As part of future work, we intend to do untestability root-causing which explains the reason as to why a fault is untestable, based on the proposed framework.

The AND/OR based framework provides the flexibility to make an arbitrary decision on any internal node in the

Design	#Gates	Fault Cov (FC) (%)		Atpg Eff (%)		Memory (MB)		Time (s)		%ΔTime
		Aztec	Concat	Aztec	Concat	Aztec	Concat	Aztec	Concat	
SD1	126K	58.63	61.39	83.88	91.80	133.83	163.34	11:51:35	8:04:50	31.87
SD2	126K	74.29	78.30	87.81	94.67	151.55	180.28	1:37:23	1:05:40	32.57
SD3	126K	77.03	80.06	88.18	95.28	170.27	201.55	9:42:10	5:37:49	41.97
SD4	126K	79.31	82.63	90.37	96.33	190	219.62	2:42:30	1:13:55	54.51
SD5	243K	74.21	76.69	95.67	97.66	337.57	584.36	6:16:36	2:02:20	67.52
SD6	130K	76	87.79	85.68	97.71	161.87	209.21	2:16:04	2:28:48	-9.36
SD7	68K	17.6	17.84	21.6	21.83	86.54	108.99	2:11:56	0:56:36	57.10

Table 3: Aztec Vs Concat on hard to detect faults

circuit and perform non-chronological backtracking based on Boolean value splitting. Non-chronological backtracking that is based on Boolean value splitting is a special case of our AND/OR reasoning framework i.e. a gate can take a value 0 **OR** 1. As part of future work, we want to study the impact of combining these two types of decision making on ATPG.

8 Acknowledgements

We would like to acknowledge the Aztec team at Intel Corporation for their help and support throughout this work. We would also like to acknowledge Vijay Durairaj, University of Utah, Salt Lake City, for implementing the SAT-based ATPG during his summer internship at Intel.

9 References

[1] S. Bommu, “CONCAT: An Efficient ATPG Engine with Conflict Learning”, Intel Invention Disclosure, # 51067, March 2006

[2] J. P. Roth, “Diagnosis of Automata failures: A calculus and a method”, *IBM Journal of Research and Development,* Vol. 10, pp. 278-291, July 1966

[3] P. Muth, “A nine-valued logic model for Test Generation”, *IEEE Transactions on Computers*, Vol. C-225, pp. 630-636, 1976

[4] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits”, *Proceedings of Fault Tolerant Computing Symposium*, 1980, pp. 145-151

[5] H. Fujiwara, “FAN: A Fanout-oriented Test Pattern Generation Algorithm”, *Proc of ISCAS*, 1985, pp. 671-674

[6] Cheng, “Split circuit model for test generation”, *Proceedings of IEEE DAC*, 1988, pp. 96-101

[7] M. Abramovici, M.A. Breuer and A.D. Friedman, “Digital Systems Testing and Testable Design”, *IEEE Press*, NJ 1990

[8] F. Breglez, “On Testability of Combinational Networks”, *Proceedings IEEE ISCAS,* 1984, pp. 221-225

[9] L.H. Goldstein, “Controllability/Observability Analysis of digital circuits”, *IEEE Transactions on Circuits and Systems,* Vol. 26, September 1979, 1984, pp. 685-693

[10] S.C. Seth, L. Pan and V.D Agrawal, “PREDICT: Probabilistic Estimation of Digital Circuit Testability, *Proc of Fault Tolerant Computing Sys,* 1985, pp. 220-225

[11] S.C. Chang, W.B. Jone and SS. Chang, “TAIR: Testability Analysis by Implication Reasoning”, *IEEE Trans on CAD,* Vol. 19, No. 1, January 2000, pp. 152-160

[12] M.H. Schulz, E. Trischler and T.M. Sarfert, “SOCRATES: A Highly Efficient Automatic Test Pattern Generation System”, *IEEE Transactions on Computer Aided Design*, Vol. 7, No. 1, 1988, pp. 126-137.

[13] I. Hamzaglou and J. Patel, “New Techniques for Deterministic Test Pattern Generation”, *Proceedings IEEE VLSI Test Symposium,* 1998, pp. 446-452

[14] W. Kunz and D.K. Pradhan, “Recursive Learning: A new implication technique for efficient solutions to CAD problems – Test, Verification and Optimization”, *IEEE Transactions of Computer Aided Design of Integrated Circuits and Systems,* 1994, pp. 1143-1158

[15] J. Giraldi and M. Bushnell, “EST: The new Frontier in Automatic Test pattern Generation”, *IEEE Transactions of Design Automation Conference,* 1990, pp. 667-673

[16] J.P. Marques Silva and K.A. Sakallah “Dynamic Search-Space Pruning Techniques in Path Sensitization”, *Proceedings of IEEE DAC,* 1994, pp. 705-711

[17] X. Lin, I. Pomeranz and S. Reddy, “Techniques for improving the efficiency of sequential test generation”, Proceedings of ICCAD, 1999, pp. 147-151

[18] H. Cox and J. Rajski, “On necessary and non-conflicting assignments in Algorithmic Test Pattern Generation”, *IEEE Transactions of Computer Aided Design of Integrated Circuits and Systems,* Vol 13, No. 4, April, 1994, pp. 515-530

[19] Chen Wang et al., “Conflict driven techniques for improving Deterministic Test Pattern Generation”, *Proceedings of IEEE Conference on Computer Aided Design,* 2002, pp. 87-93

[20] T. Larrabee, “Test pattern generation using Boolean Satisfiability”, *IEEE Transactions of Computer Aided Design,* Vol. 11, January 1992, pp. 4-15

[21] J.P. Marques Silva and K.A. Sakallah, “GRASP – A new search algorithm for satisfiability”, *Proceedings of IEEE International Conference on Computer Aided Design,* 1996, pp. 220-227

[22] Moskewicz et al., “Engineering an efficient SAT solver”, *Proceedings of IEEE Design Automation Conference,* 2001, pp. 530-535

[23] S. Chakradhar et al., “A transitive closure algorithm for test generation”, *IEEE transactions on Computer Aided Design”,* Vol. 12, July 1993, 1015-1028

[24] P. Stephan et al., “Combinational test generation using satisfiability”, *IEEE transactions on Computer Aided Design”,* Vol. 15, September 1996, pp. 1167-1176

[25] Gizdarski and Fujiwara, “SPIRIT: A highly combinational Test Generation Algorithm”, *IEEE transactions on Computer Aided Design of Integrated Circuits and Systems,* Vol. 21, No. 12, December 2002, pp. 1446-1458

[26] Lu et al., “An Efficient Sequential SAT solver with improved search strategies”, *Proccedings of IEEE Conference on DATE,* 2005, pp. 1102-1107

[27] M.K. Ganai et al., “Combining strengths of circuit-based and CNF-based algorithms for high performance SAT solver”, *Proceedings of IEEE Design Automation Conference,* 2002, pp. 747-750

[28] L. Zhang et al., “Efficient Conflict Driven Learning in a Boolean Satisfiability Solver”, *Proceedings of IEEE International Conference on Computer Aided Design,* 2001, pp. 279-285