

Fault Diagnosis and Logic Debugging Using Boolean Satisfiability

Andreas Veneris
University of Toronto
Departments of ECE and CS
Toronto, ON M5S 3G4
veneris@eecg.toronto.edu

Abstract

Recent advances in Boolean satisfiability have made it attractive to solve many digital VLSI design problems such as verification and test generation. Fault diagnosis and logic debugging have not been addressed by existing satisfiability-based solutions. This paper attempts to bridge this gap by proposing a model-free satisfiability-based solution to these problems. The proposed formulation is intuitive and easy to implement. It shows that satisfiability captures significant problem characteristics and it offers different trade-offs. It also provides new opportunities for satisfiability-based diagnosis tools and diagnosis-specific satisfiability algorithms. Theory and experiments validate the claims and demonstrate its potential.

1 Introduction

Recent years have seen an increased use of Boolean Satisfiability (SAT) based tools in the design cycle. Design verification and model checking [4][5], test generation [6], optimization [10] and physical design [12], among others, have been successfully tackled with SAT-based solutions. This is due to recent advances in SAT solvers [7] [9] that make them efficient tools for these problems.

Although SAT-based solutions have tackled many circuit design problems, *design diagnosis* has not yet been addressed in existing literature. Given an erroneous design, a specification and a set of input test vectors, diagnosis identifies malfunctioning portions of the design. Diagnosis is an integral process to improve the design cycle, increase manufacture yield and shorten the time-to-market window [2] [3].

Depending on the stage of the design cycle, shown in Fig. 1, and the type of malfunction (“soft” or “hard”), diagnosis is required during design error diagnosis (logic debugging) and during fault diagnosis. *Design error diagnosis* occurs in early stages of the design

cycle where the specification is some HDL (or RTL) description and the design is a logic netlist. Malfunctions are caused by specification changes, bugs in automated tools or the human factor [1]. Logic debugging identifies lines and corrections in the erroneous netlist that correct it according to a specification. *Fault diagnosis* takes place when the fabricated chip fails testing. Given a faulty chip and a netlist, fault diagnosis identifies locations in the correct netlist by injecting faults into it until the netlist emulates the behavior of the faulty chip. Since both problems have similar goals, we describe this work in terms of fault diagnosis unless otherwise stated.

It is notable that diagnosis is an inherently difficult problem because the solution (search) space grows exponentially with the number of circuit lines, the number of faults and the various fault models : $diagnosis\ space = (\#\ ckt\ lines)^{(\#\ errors)}$. This is because the specification (HDL or the failing chip) is treated as a “black box” controllable at the primary inputs and observable at the primary outputs (Fig. 2). Due to this complexity, development of efficient diagnosis tools remains a challenging task.

Motivated by these observations, we present a *SAT-based solution* to design diagnosis of multiple faults. The formulation is intuitive, straightforward to implement and decouples diagnosis from fault modeling. *Model-free* diagnosis is a desirable characteristic for modern devices where fault effects may have non-deterministic (unmodeled) behavior [3].

In this work we do not develop a SAT solver but we propose a SAT-based solution to fault diagnosis and we use existing solvers to solve it. We argue that SAT naturally captures many essential characteristics of diagnosis, we examine different implementation trade-offs and suggest heuristics to guide a SAT solver towards an efficient solution. To the best of our knowledge, this is the first work to examine design diagnosis using SAT. Experiments with multiple faults demonstrate the effi-

ciency and practicality of the approach.

This paper is organized as follows. Section 2 contains background information and definitions. Section 3 describes the proposed SAT-based formulation and its characteristics. Section 4 contains experiments and the last Section concludes this work.

2 Background

Traditionally, diagnosis techniques are classified as *cause-effect* or *effect-cause* techniques [2]. Cause-effect analysis usually compiles fault dictionaries. Given a failing chip and a set of vectors v_1, v_2, \dots, v_k from the tester, the chip responses are matched with those in the dictionary to return set of potential faults for each vector. Effect-cause analysis does not use fault dictionaries but simulates input vectors and applies different techniques to identify candidate faults.

In both cases, sets of candidate faults F_1, F_2, \dots, F_k are returned. When each F_i is injected in the netlist, it explains the (faulty or non-faulty) behavior of test vector v_i alone. These sets are later *intersected* $F = F_1 \cap F_2 \cap \dots \cap F_k$ to return set F of faults that explains the chip behavior for *all* vectors v_1, v_2, \dots, v_k .

The quality of diagnosis is related to its *resolution*, that is, its ability to return in F the line(s) where fault(s) reside. Due to fault equivalence [2], a solution may not be unique. Ideally, a solution contains only the actual and equivalent fault sites, because it is easier for the designer to probe these sites.

In this work, we consider combinational circuits with primitives AND, OR, NOT, NAND, NOR, XOR and XNOR gates and full-scan sequential circuits with a fault-free scan-chain. We use Conjunctive Normal Form (CNF) SAT instances expressed as a logical AND (\cdot) of *clauses*, each of which is the OR ($+$) of one or more literals. A literal is an instance of a variable x or its negation x' . We use the procedure in [6] to translate logic circuits into CNF form. Given a CNF formula, a SAT solver finds a variable assignment that satisfies the formula or it proves that the formula cannot be satisfied.

Without loss of generality, we describe our algorithms on circuits with m primary inputs $X = x_1, x_2, \dots, x_m$ and a single primary output $y = f(x_1, x_2, \dots, x_m) = f(X)$. The method is easily generalized to multiple output circuits. We use the names $L = \{l_1, l_2, \dots, l_n\}$ to represent internal circuit lines including stems and branches. The method in Section 3 adds circuitry to the original circuit. This new hardware requires two extra lines per original circuit line and we use $S = \{s_1, s_2, \dots, s_n\}$ and $W = \{w_1, w_2, \dots, w_n\}$ to label these lines.

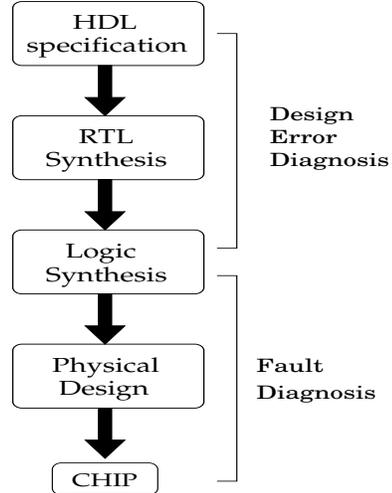


Figure 1: Digital VLSI design flow

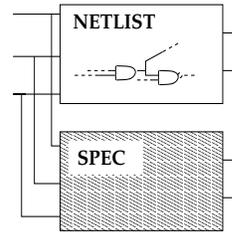


Figure 2: Fault diagnosis and logic debugging

In this presentation, variables for all circuit lines x_i, l_i, w_i and y are defined to model circuit constraints *under* simulation of vector v_j . To avoid confusion, we use the notation x_i^j, l_i^j, w_i^j and y^j for these variables and X^j, L^j and W^j for the respective sets of variables. Under this notation, superscript j matches the index of simulated test vector v_j . The notation $S = \{s_1, s_2, \dots, s_n\}$ is used to indicate both variable and line names. Variables for lines S are common to *all* test vectors.

3 SAT-based Design Diagnosis

Given a logic netlist and a set of vectors v_1, v_2, \dots, v_k , the SAT-based algorithm introduces new logic and compiles a CNF formula Φ for this new circuit to model vector constraints. Formula Φ has *two components*.

The *first component* is the conjunction of k CNF formulas $C^j(L^j, W^j, X^j, y^j, S)$, $1 \leq j \leq k$. Each such C^j enforces vector v_j constraints in the logic netlist and potential fault sites. This is done with circuitry added to the design. The *second component* $E_N(S)$ describes constraints for the number N of injected faults.

These constraints are also coded in the circuit with hardware and they are later translated to CNF. N is user-specified and it states that the design is corrupted with N faults.

The complete formula Φ is expressed as:

$$\Phi = E_N(S) \cdot \prod_{j=1}^k C^j(L^j, W^j, X^j, y^j, S)$$

Intuitively, $\prod_{j=1}^k C^j(L^j, W^j, X^j, y^j, S)$ requires that the candidate set of faults satisfies every C^j constraint for all vectors v_j . In other words, faults are *intersected* for all vectors as in traditional diagnosis. We now describe how each component of Φ is formed with theory and examples.

Component 1: This is comprised of k CNF formulas C^j to model the circuit and fault constraints for vector v_j . The circuit is modified to reflect the presence of faults at various circuit lines. To model the presence of a fault on line l_i , a multiplexer with select line S_i is attached to this line. This multiplexer is later translated into CNF format and added to the formula.

Consider the circuit in Fig. 3(a), for example. The presence of a fault on line $l = g \rightarrow h$ can be represented by a multiplexer with select line s , as shown in Fig. 3(b) and explained in [11]. The first input of the multiplexer is connected to the output of gate g and the second input of the multiplexer is connected to a new line w to model the potential fault. The output of the multiplexer is connected to the original output of g . Observe that the functionality of the original or faulty circuit is selected when the value of the select line is 0 or 1, respectively.

The CNF for the multiplexer logic is given in Fig. 3(c). It can be seen that only 4 clauses are required. Hence, the CNF formula for the complete circuit in Fig. 3(b) is $C = (x_1 + l') \cdot (x_2 + l') \cdot (x'_1 + x'_2 + l) \cdot (s + l' + z) \cdot (s + l + z') \cdot (s' + w' + z) \cdot (s' + w + z') \cdot (x_3 + y) \cdot (z + y) \cdot (x'_3 + z' + y')$.

Once multiplexers are introduced at every line, the new circuit is translated into CNF. To get C^j from this CNF formula, we insert clauses that represent input/output constraints of the test vectors v_j . This can be done with a set of unit-literal clauses for the set of primary input variables x_1, x_2, \dots, x_m and primary output y . These literals agree with the respective logic values of the vector v_j ; that is, if v_j assigns a logic 1 (0) to input x_j then x_i^j ($x_i^{j'}$) appears in the formula and so on.

Example: Recall the circuit in Fig. 3(a) and assume there is a single stuck-at 1 fault on line l . The input

test vector $v = (x_1, x_2, x_3) = (1, 0, 1)$ detects the fault: a logic 1 appears at the output of the good circuit, while a logic 0 appears at the output of the faulty one. The construction requires unit-literal clauses x_1, x_2, x_3 and y' to be added to C . Hence, the final CNF formula for vector v is $C^v = C \cdot x_1 \cdot x'_2 \cdot x_3 \cdot y'$.

This process is repeated for every test vector v_j , $j = 1 \dots k$ to get CNFs $C^j(L^j, W^j, X^j, y^j, S)$. Note that each such formula requires a new set of variables (and literals) for primary inputs (X^j), primary outputs (y^j), internal circuit lines (L^j) and fault sites (W^j). This is because every input test vector may translate into a different set of constraints for circuit lines and fault locations. However, only one set of select line $S = s_1, s_2, \dots, s_n$ variables is used because the fault(s) of a solution must satisfy all the vector constraints simultaneously. The second component, described next, constrains the cardinality N of these faults.

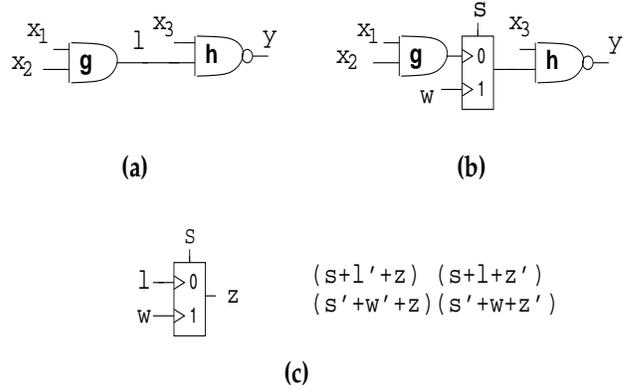


Figure 3: Circuit multiplexer construction C

Component 2: The second component attaches additional hardware to the circuit above. This logic translates into constraints $E_N(S)$ that request a solution with at most N faults. When this new circuit is translated into CNF, we obtain formula Φ . We describe the idea for the single fault case ($E_1(S)$) first. Later, we generalize for multiple faults and discuss trade-offs.

Example: Consider the formula C^v computed by the first component. This formula models the circuit in Fig. 3(b) under simulation of test vector $v = (1, 0, 1)$. Assume s (multiplexer select line) is introduced as an additional unit-literal clause so that the formula becomes $C^v = C \cdot x_1 \cdot x'_2 \cdot x_3 \cdot y' \cdot s$. Given this new C^v , a SAT solver will attempt to find a satisfying variable assignment for the circuit lines *and* the variable w so that the circuit emulates the faulty chip behavior for vector v . The multiplexer will be forced to select line w

and the solver will return $w = 1$ to indicate a stuck-at 1 fault on line l .

The general procedure for single faults is an extension of the one in the example. Given variables for select lines $S = s_1, s_2, \dots, s_n$, we need to add the component $E_1(S)$ to indicate that one, and only one select line may be set to logic 1 at any time. This is done with the following:

$$E_1(S) = (s_1 + s_2 + \dots + s_n) \cdot \prod_{\substack{i=1 \dots n-1, \\ j=i+1 \dots n}} (s'_i + s'_j)$$

The left part requires that at least one select line be set to logic 1, and the right part causes $E_1(S)$ to become unsatisfied if more than one select line is set to 1. Clearly, the set of new clauses introduced by $E_1(S)$ is $O(n^2)$. This idea can be extended to multiple errors. For example, it can be shown that

$$E_2(S) = (s_1 + s_2 + \dots + s_n) \cdot \prod_{\substack{i=1 \dots n-2, \\ j=i+1 \dots n-1, \\ p=j+1 \dots n}} (s'_i + s'_j + s'_p)$$

requires the SAT solver to search for one or two faults etc. Although this formulation of $E_N(S)$ may be practical for single faults, it requires an exponential number of clauses to be added *explicitly* to the formula. For example, $E_2(S)$ for a circuit with 10^3 lines requires 10^9 new clauses to be included.

To overcome this exponential explosion of space requirements in the multiple fault case, we introduce the circuit the hardware shown in Fig. 4. This hardware acts as a counter, forcing the SAT solver to “enumerate” sets of N fault sites. In the figure, thick lines indicate buses of $O(\log n)$ bit-width ($N \leq n$) and all other lines represent single bit buses. The hardware performs a bitwise addition of the multiplexer select lines $S = s_1, s_2, \dots, s_n$ and compares the result to the user-defined number of faults N . The output of the comparator is “forced” to logic 1 with a unit-literal clause so that the bitwise addition of the members of S is always equal to N in the comparator. As with the select lines themselves, the variables introduced for this hardware are common to all vectors v_j .

It can be shown that the number of CNF clauses introduced by $E_N(S)$ with this hardware construction remains linear $O(n)$. We omit proof of the claim due to lack of space. Intuitively, this *implicit* hardware representation for $E_N(S)$ provides a *trade-off* between time and space. In the section that follows, we argue that modern SAT-solvers take advantage of this trade off in practice to avoid an exponential explosion in the time

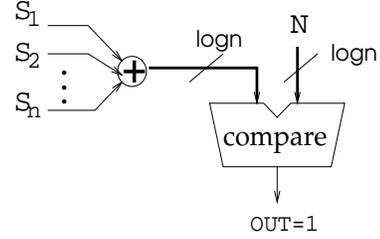


Figure 4: Hardware for multiple errors

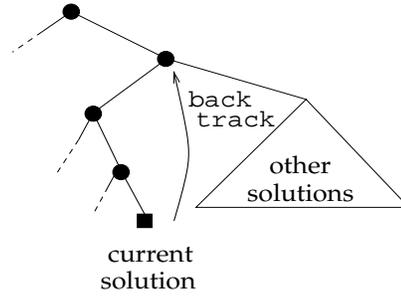


Figure 5: Implementation heuristics

domain. Experiments in Section 4 confirm this observation.

Implementation Details: As explained, a multiplexer requires 4 additional clauses, and the counter construction in Fig. 4 is done with $O(n)$ clauses. Therefore, *space requirements* for Φ are *linear* $O(nk)$ in both the number of circuit lines and the number of vectors. In the remainder of this paper, we discuss *time requirements* and explain why the proposed SAT-based formulation performs model-free diagnosis.

Modern SAT-solvers [7] [9] are enriched with clause-learning and backtracking techniques to help search and prune the solution space. To take advantage of these techniques, the SAT solver is modified as follows.

For every multiplexer with select line s_i and inputs l_i and w_i , clause $(s_i + w_i^{j'})$ is added for vector v_j to denote the logic implication $s_i' \rightarrow w_i^{j'}$. This has the desirable effect that when fault on line l_i is not selected ($s_i = 0$), then the value on w_i is immediately set to logic 0 to prevent unnecessary branching of the SAT solver on the value of w_i . Additionally, as soon as the solution of fault sites $s_{i_1}, s_{i_2}, \dots, s_{i_N}$ is returned, the SAT-solver does not reset and start to search for another solution from scratch. Instead, the clause $(s'_{i_1} + s'_{i_2} + \dots + s'_{i_N})$ is immediately added as a *learned clause*.

This is illustrated in Fig. 5 where dotted lines indicate explored portion of the solution space. Upon discovery of a solution, the tool backtracks and may reuse part of the past computation to identify other so-

lution(s) or return unsatisfiability (no other solutions). This is useful in fault diagnosis where all actual and equivalent solutions need be probed by the test engineer. Experiments show that this heuristic helps a SAT solver tackle the run-time complexity when it searches for all solutions [8]. In debugging, the tool usually exits as soon as it finds the first solution.

The SAT-based formulation does not make any assumption on the logic value of the fault for each vector v_j . Therefore, it provides a model-free approach to diagnosis. This is a desirable characteristic because it may capture faults with “non-deterministic” behavior [3]. However, it is interesting to argue on the logic assignments to variables $w_{i_1}^j, w_{i_2}^j, \dots, w_{i_N}^j$ on circuit lines $l_{i_1}, l_{i_2}, \dots, l_{i_N}$ of a solution for all vectors v_j .

As explained, these logic line assignments are required to guarantee that the netlist emulates the behavior of the specification for v_j . The test engineer may use these values to determine the behavior of the fault and perform fault modeling [2]. Because of these characteristics, we can conclude that SAT provides an attractive platform for fault diagnosis and logic debugging.

4 Experiments

In this Section we present experiments for a prototype tool implemented in the C programming language. Hardware construction and heuristics are embedded in the code of the SAT-solver described in [7]. Experiments are conducted for single and double stuck-at faults in the ISCAS’85 benchmark circuits. We use the original versions of the benchmark circuits where *C7552* has 7552 lines, *C432* has 432 lines etc. These versions contain redundancies and they are harder to diagnose. The type and location of the faults are selected at random. We run experiments on a SUN Blade 100 workstation with 256MB of memory. Ten experiments are performed for each circuit and for each fault case. Average values are reported in the next paragraphs and run-times are in seconds.

Table 1 contains results on single stuck-at faults and Table 2 shows information in a similar manner for double faults. The first column of each table has the circuit name and the second column contains the number k of test vectors used for diagnosis. This set contains mainly vectors with failing responses. Test vector generation is not the subject of this work [6].

The third column of each table has the initial number of clauses of Φ before learned clauses are added. These numbers confirm that memory requirements are linear to circuit size and number of vectors. For example,

Table 1: Single stuck-at faults

ckt name	# of vectors	# of clauses	# fault sites	CPU (sec)	
				one	all
C432	30	48,181	5.6	0.4	0.1
C499	30	142,314	9.4	0.2	0.2
C880	30	108,112	11.4	1.0	0.3
C1355	30	141,388	6.4	1.9	0.6
C1908	30	102,322	3.1	3.1	1.2
C2670	60	420,033	7.4	4.0	1.7
C3540	60	735,345	6.2	15.0	9.1
C5315	60	488,345	12.2	29.0	8.8
C6288	60	1,654,667	2.7	104.1	200.8
C7552	60	1,230,687	6.1	30.1	17.6

C432 requires approximately half the number of clauses of *C880* because it has nearly half the number of lines. Equivalently, the CNF sizes of circuits with single faults are half of that for double faults because diagnosis uses half the number of test vectors.

The number of fault sites (actual and equivalent) returned is found in column 4. The next columns have *total* CPU times. CPU time per fault can be found if we add these columns and divide with the number of faults. The average CPU time to return the first solution is found in column 5. Once the first solution is found, column 6 contains the average CPU time to return *subsequent* solutions and/or to prove unsatisfiability. In most cases, SAT is very efficient for diagnosis.

We observe that the SAT-solver spends more time to return the first solution than all others. The CPU run-times in the last two columns confirm the intuition behind Fig. 5 and they suggest that the added clauses allow the computation performed for the first solution to be reused by the tool to find other solutions. The benefit of these heuristics (Section 3) is also depicted in Fig. 6. This figure shows the SAT solver run-time for single faults when none, one or both of heuristics are employed. Recall that the first heuristic requires variable w_i^j on line l_i immediately to assume a logic 0 once s_i is not selected for vector v_j . The second heuristic backtracks once a solution is found to reuse past computation and return more solutions. Run-times indicate that the added clauses allow the SAT solver to prune the solution space. For *C3540*, for instance, the speed up is dramatic.

Experiments demonstrate the effectiveness, flexibility and practicality of the SAT-based solution to design diagnosis. In the future, we plan develop diagnosis-specific satisfiability algorithms to improve performance.

Table 2: Double stuck-at faults

ckt name	# of vectors	# of clauses	# fault sites	CPU (sec)	
				one	all
C432	60	96,249	13.2	6.7	0.4
C499	60	288,923	23.2	42.1	4.5
C880	60	215,076	17.2	13.1	1.5
C1355	60	286,583	12.4	51.0	2.6
C1908	60	201,556	28.8	20.8	7.4
C2670	120	882,143	24.5	72.7	11.3
C3540	120	987,798	3.3	188.5	102.8
C5315	120	1,695,180	24.3	308.6	15.8
C6288	120	3,240,767	2.2	1011.8	1712.2
C7552	120	2,410,767	3.7	432.1	555.8

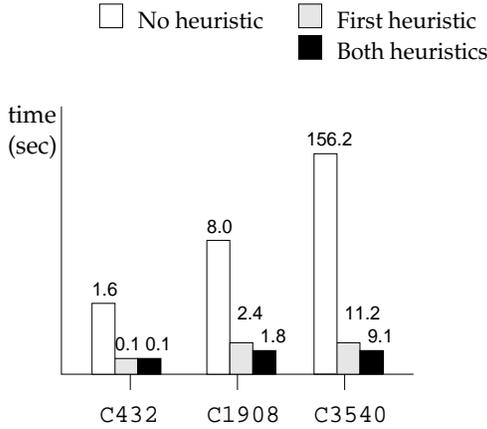


Figure 6: Performance speed up

5 Conclusions

A satisfiability-based solution to multiple fault diagnosis and logic debugging is presented. The method is intuitive and practical within an industrial environment. Theoretical and experimental results indicate that Boolean satisfiability provides an efficient solution to design diagnosis. This gives new opportunities for satisfiability-based diagnosis tools and diagnosis-specific satisfiability algorithms.

Acknowledgments

The author thanks Dr. Anastasios Viglas and Alexander Smith for many technical contributions and insights. He also acknowledges Prof. Spyros Tragoudas for his encouragement in early steps of the work.

References

- [1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification Via Test Generation," in *IEEE Trans. on CAD*, vol. 7, pp. 138–148, Jan. 1988.
- [2] M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital Systems Testing and Testable Design," *IEEE Press*, 1991.
- [3] R. C. Aitken, "Modeling the Unmodelable: Algorithmic Fault Diagnosis," in *IEEE Design and Test of Computers*, pp. 98–103, July–Sept. 1997.
- [4] P. Bjesse, T. Leonard and A. Mokkedem, "Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers," in *Proc. of Int'l Conf. on CAV, Lecture Notes in Computer Science, Springer-Verlag*, vol. 2102, pp. 454–464, July 2001.
- [5] E. Goldberg, M. Prasad and R. Brayton, "Using SAT for Combinational Equivalence Checking," in *Proc. of IEEE DATE*, pp. 114–121, 2001.
- [6] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," in *IEEE Trans. on CAD*, vol. 11, no. 1, pp. 4–15, Jan. 1992.
- [7] M.H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of DAC*, pp. 530–535, 2001.
- [8] M. R. Prasad, "Propositional Satisfiability Algorithms in EDA Applications," *Ph.D. Thesis, University of California, Berkeley*, 2001.
- [9] J. P. M.-Silva and K. A. Sakallah, "GRASP – A Search Algorithm for Propositional Satisfiability," in *IEEE Trans. on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.
- [10] P. Tafertshofer, A. Ganz and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists," in *Proc. of ICCAD*, pp. 648–657, 1997.
- [11] A. Veneris and M. S. Abadir, "Design Rewiring Using ATPG," in *Proc. IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1469–1479, Dec. 2002.
- [12] R. G. Wood and R. A. Rutenbar, "FPGA Routing and Routability Estimation via Boolean Satisfiability," in *IEEE Trans. on VLSI Systems*, vol. 6, no. 2, pp. 222–231, June 1998.