On Applying Incremental Satisfiability to Delay Fault Testing

Joonyoung Kim, Jesse Whittemore, João P. Marques-Silva[a] and Karem Sakallah
University of Michigan {jykim, jwhitte, karem}@umich.edu
[a] IST/INESC, Cadence European Laboratories jpms@inesc.pt

Abstract

The Boolean satisfiability problem (SAT) has various applications in electronic design automation (EDA) fields such as testing, timing analysis and logic verification. SAT has been typically applied to EDA as follows: 1) formulation of the given problem as a SAT instance 2) solution of the SAT instance. In this paper, we present a method to simultaneously solve several closely related SAT instances using incremental satisfiability (ISAT). In ISAT, the decision sequence made for a "prefix" function is used to solve another set of functions which have a number of new constraints (extensions) added to the prefix function. Our experiments show that we can achieve significant gains in total runtime when we use this methodology as opposed to resetting the decision sequences and solving each instance from scratch. Application of ISAT to delay fault testing is presented by formulating incremental path sensitization as an ISAT problem. Non-robust tests for the combinational portion of ISCAS 89 circuits are generated using this method.

I. Introduction

Many electronic design automation (EDA) problems can be cast as instances of Boolean satisfiability (SAT): given a Boolean function $\varphi(X):\{0,1\}^n \rightarrow \{0,1\}$, find an assignment X^* such that $\varphi(X^*) = 1$ or prove that no such assignment exists. The function $\varphi(X)$ is typically expressed in conjunctive normal form (CNF) and the problem is usually solved by a backtracking search algorithm that systematically explores the *n*-dimensional Boolean space of the variables. Even though it is well known that SAT is NP-complete, recent advances in this field, e.g. [1][13][14], have made it possible to apply SAT to problems of considerable size.

Incremental satisfiability was first considered in [7] but its application was limited to solving one large problem by partitioning it into *m* partitions where *m* is the number of clauses and solving them incrementally by adding one clause at a time. In this paper we propose a modification to the incremental satisfiability problem suggested in [7], and a corresponding modification to the search algorithm, to address a situation that frequently arises in many application domains. The *incremental satisfiability* (ISAT) problem is: determine the satisfiability of each function $\varphi_i(X)$ in a set of *m* Boolean functions $\Phi = \{\varphi_i(X) \mid \varphi_i(X) = \varphi_P(X) \cdot \varphi_{Si}(X)\}$ that share a common *prefix* function $\varphi_P(X)$ but have different *suffix* functions $\varphi_{Si}(X)$. We will refer to each such function $\varphi_i(X)$ as an *extension* of the common prefix $\varphi_P(X)$. This problem can be obviously cast and solved as *m* independent SAT problems. Significant computational savings can be realized, however, by solving them "together." Specifically, a solution is found for the common prefix, which is then extended or slightly modified to obtain solutions to each of the *m* extensions. The maximum benefit from such an approach accrues when $\varphi_P(X)$ is unsatisfiable since that obviates the need to even consider the suffixes to prove that all of the *m* extensions are unsatisfiable. The satisfiability of $\varphi_P(X)$, on the other hand, is a necessary but not sufficient condition for the satisfiability of its *m* extensions. The potential savings in this case are harder to predict. In particular, significant savings are still possible if the solutions to each of the *m* extensions can be found by *augmenting* the partial solution obtained for the prefix with assignments to previously unassigned variables. A more likely scenario, however, is that the solution to at least some of the extensions can only be obtained by *reversing* existing assignments in the solution found for the prefix. Fortunately, in the EDA application domain, the size of the prefix is usually much larger than the sizes of each of the suffixes. This means that the performance gain obtained by incremental SAT can be larger than the performance loss from reversing current assignments.

Boolean satisfiability has been widely used for both stuck-at fault testing [9] and delay fault testing [2] of combinational circuits. In this paper we will apply ISAT to prove the untestability of non-robust delay faults in logic circuits.

Test pattern generation for path delay fault has been researched heavily recently. Various classifications of delay faults (e.g., robust, non-robust, validatable non-robust etc.) exist. Although robust test is preferred to non-robust test in general, there have been some research results that prove the importance of non-robust test [4][12].

It should be noted that the objective of this paper is not to advocate a certain kind of delay fault model. Non-robust delay fault generation was chosen in this paper since it allows for straightforward application of ISAT to EDA problems.

This paper is organized as follows. In Section II incremental satisfiability is introduced. The application of ISAT to delay fault testing is presented in Section III. Experimental results and conclusions are presented in Section IV and Section V, respectively.

II. Incremental Satisfiability

In this paper we consider Boolean functions represented in Conjunctive Normal Form (CNF). A *literal* is an occurrence of a variable or its complement. A *clause* is a disjunction (OR) of literals. A CNF formula is a conjunction (AND) of clauses.

A CNF formula is said to be *satisfiable* when there is at least one truth assignment to its variables that makes all clauses equal to 1. A CNF formula is said to be *unsatisfiable* when no such assignment exists.

Let us consider a case when the given Boolean function $f_p(x_1, x_2, \ldots, x_n)$ is found to be satisfiable. If a set of additional clauses which represent the function f_S is added to f_P, what can we say about the satisfiability of the overall function $f = f_p \cdot f_S$? Note that the satisfiability of f_P is a necessary, but not sufficient condition for the satisfiability of f. Instead of "resetting" all the decisions made in deriving the solution for f_P, we can resume the search by updating the status of the clauses in f_S against the assignments that were made for the problem f_P for the overlapping portion of the support set.

When we update the new clauses against the current assignment, there is no need to reverse current assignments unless some new clause becomes unsatisfied as the following example illustrates.

Example 1

Consider a prefix boolean function $f_P(x_1, x_2, x_3, x_4) = (\overline{x_1} + x_3)(x_2 + \overline{x_3})(x_1 + \overline{x_2} + x_3)$. A possible decision tree of the satisfying assignment $\{x_1 = 0, x_2 = 1, x_3 = 1\}$ for this problem is shown in Figure 1(a). Now let us add the clauses $f_{S1}(x_1, x_2, x_3, x_4) = (\overline{x_2} + x_4)(x_3 + \overline{x_4})$ and solve a new problem $f_1 = f_P \cdot f_{S1}$. The current assignment can be used to update the status of the new clauses. As a result, the first clause of f_{S1}, $(\overline{x_2} + x_4)$ becomes a unit clause (an unresolved clause with only one free literal) and the second clause $(x_3 + \overline{x_4})$ becomes satisfied. The resulting decision sequence after satisfying the unit clause is shown in Figure 1(b).

Now let us consider adding a different set of clauses to the prefix function. Let $f_{S2}(x_1, x_2, x_3, x_4) = (\overline{x_2} + \overline{x_3})(x_3 + \overline{x_4})$ and $f_2 = f_P \cdot f_{S2}$. The first clause $(\overline{x_2} + \overline{x_3})$ becomes unsatisfied when we update the clause with the current assignment. The conflict analysis procedure [13] will be triggered and generate a conflict induced clause $(x_1 + \overline{x_2})$, which "asserts" the value of x_2 to be 0 under the assignment $\{x_1 = 0\}$. The decision sequence for this problem is shown in Figure 1(c) with assignment $\{x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0\}$.

Finally let us consider adding $f_{S3}(x_1, x_2, x_3, x_4) = (\overline{x_2} + \overline{x_3})(x_1 + x_3)$ to obtain the function $f_3 = f_P \cdot f_{S3}$ (Figure 1 (d)). Not only does the addition of function f_{S3} conflict with current assignments, it makes the overall function f_3 unsatisfiable under any assignment. ■

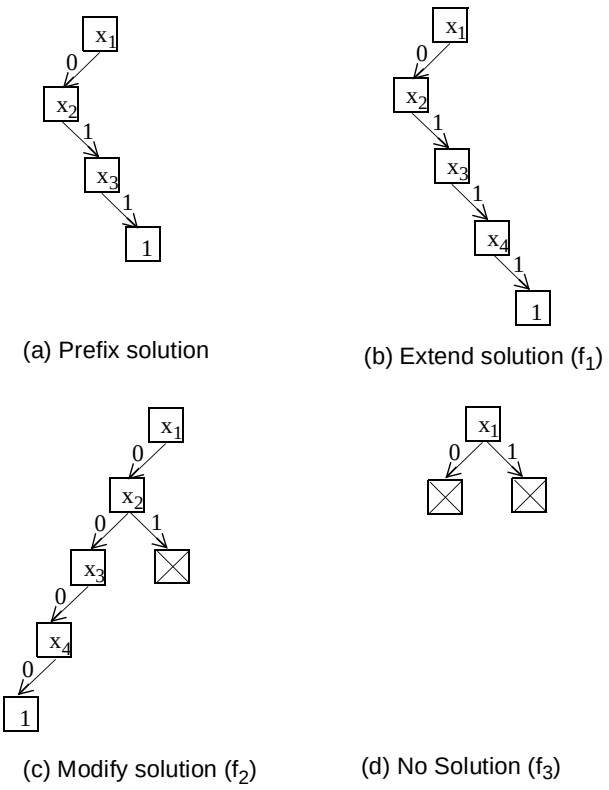


Figure 1: Incremental Search example

III. ISAT Applied to Delay Fault Testing

Delay fault testing is performed after fabrication of an integrated circuit. There are two widely used fault models, gate delay faults and path delay faults. We will consider the path delay fault model [11] [3] in this paper.

A path delay fault models distributed fault effects on a given path which causes the delay of the path to exceed a specified limit. For each structural path, two path delay faults can be considered, rising and falling transition at the output of the path.

To detect a delay fault, it is common to apply a pair of vectors $< v_1, v_2 >$ at the inputs of the circuit and sample the output of the circuit after one clock period. The interval between v_1 and v_2 must be long enough so that all signals in the circuit can stabilize.

A test for a given delay fault is called robust if the test can detect the given fault independent of the presence of other delay faults in the circuit. A test is called non-robust if the given fault can be masked by the presence of other delay faults in the circuit.

Our experimental results are based on the non-robust delay fault testing model. Note that the side input condition of v_2 for non-robust faults is exactly the same as the static sensitization criterion in timing analysis. Hence if we have an efficient way of checking static sensitizability, it can be readily used as a test pattern generator for non-robust faults.

The approach used in [2] extracts a fan-in cone for each primary output and finds a test for each path to that primary

gate type	consistency function for z=f(x,y)
AND	$(x + \bar{z})(y + \bar{z})(\bar{x} + \bar{y} + z)$
OR	$(\bar{x} + z)(\bar{y} + z)(x + y + \bar{z})$
NAND	$(x + z)(y + z)(\bar{x} + \bar{y} + \bar{z})$
NOR	$(\bar{x} + \bar{z})(\bar{y} + \bar{z})(x + y + z)$

TABLE I: Consistency functions of basic gates

output. This improves on the simplistic path-by-path approach since extraction is performed only once for each primary output. However, both the size of each SAT problem and the number of paths for each primary output can become very large for most practical circuits.

Instead of explicitly enumerating all paths, we can use incremental satisfiability to prune away large portions of untestable faults by applying ISAT to incremental path sensitization as proposed in [5]. Consider a partial path $P = (g_0, l_0, \ldots l_{i-1}, g_i)$ that starts from a primary input g_0. If the path P is untestable for delay faults, then all other paths that have P as a prefix can be removed from consideration since these faults are untestable as well. We can use depth first search when choosing paths, backtracking when an untestable fault is identified. The algorithm is illustrated in Figure 2. We generate *consistency functions* which enforce consistent assignments to the inputs and output of each gate according to its functionality.

For a gate z which realizes a function $f(x_1, \ldots, x_n)$, we can generate the consistency function by deriving a CNF representation of $\overline{z \oplus f}$.

Example 2

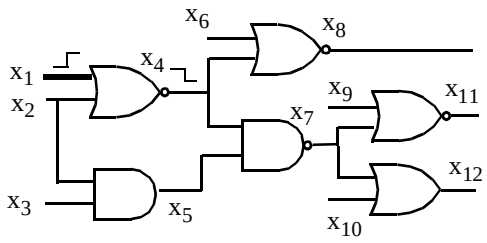
Consider a two input AND gate $z = xy$. The consistency function for this gate can be derived as follows.

$$\begin{aligned}&\overline{z \oplus xy}\\ &= z\overline{xy} + \bar{z}(xy)\\ &= (\bar{z} + xy)(z + \bar{x} + \bar{y})\\ &= (\bar{z} + x)(\bar{z} + y)(z + \bar{x} + \bar{y})\end{aligned}$$

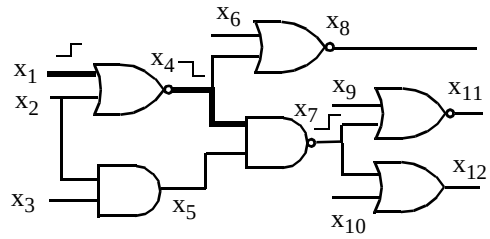
```
incremental_search (){
    for each PI x
        recurse(x, rising);
        recurse(x, falling);
}


recurse(x, direction){
    for each fanout(x) y
    if y is PO
        sensitizable path found;
    else
        if (test_sensitization() == SUCCESS)
            if x is non-inverting
                recurse(y, direction);
            else
                recurse(y, inverse(direction));
        else
            return;
}
```

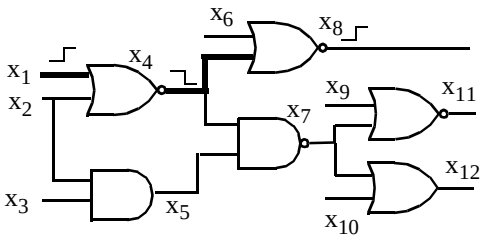
Figure 2: Outline of incremental search



(a) Path $P_p = (x_1, x_4)$



(b) Path $P_1 = (P_p, x_7) = (x_1, x_4, x_7)$



(c) Path $P_2 = (P_p, x_8) = (x_1, x_4, x_8)$

Figure 3: Example of partial path sensitization

Consistency functions of simple two input gates are shown in Table I. Side input constraints are then added to enforce the static sensitization condition which requires side inputs to assume non-controlling values. Side input conditions of complex gates can be generated using the Boolean difference of the gate function with respect to the on-path input.

The next example illustrates the application of ISAT to the path sensitization problem.

Example 3

Consider the path $P_p = (x_1, x_4)$ with a rising input at x_1 in Figure 3. The function $f(P_p)$ that captures non-robust delay fault testability of P_p is:

$$f(P_p) = (x_1)(\bar{x_1} + \bar{x_4})(\bar{x_2} + \bar{x_4})(x_1 + x_2 + x_4)(\bar{x_4})(\bar{x_2})$$

Note that these clauses consist of

- the consistency function of x_4:

$$(\bar{x_1} + \bar{x_4})(\bar{x_2} + \bar{x_4})(x_1 + x_2 + x_4)$$

- on-path transition constraints: $(x_1)(\bar{x_4})$
- side input conditions: $(\bar{x_2})$

$f(P_p)$ can be satisfied by $\{x_1 = 1, x_2 = 0, x_4 = 0\}$. Consider next path $P_1 = (P_p, x_7) = (x_1, x_4, x_7)$. Five additional clauses must be added to $f(P_p)$ yielding:

				Incremental method					Explicit path enumeration				
Circuit	# testable faults	# faults	% testable faults	# SAT calls	avg # of clauses per call	avg # of variables per call	Time(s)	Time (ms)/ sat call	# SAT calls	avg # of clauses per call	avg # of variables per call	Time(s)	Time (ms)/ sat call
s344	654	710	92.1%	3,620	78.6	31.9	0.6	0.17	710	92.5	37.5	0.8	1.17
s349	656	730	89.9%	3,640	78.4	31.8	0.6	0.18	730	91.3	36.9	0.9	1.18
s382	734	800	91.8%	2,235	45.1	20.2	0.5	0.23	800	62.2	27.0	0.8	0.99
s386	414	414	100.0%	2,646	53.4	23.0	0.6	0.21	414	74.8	31.6	0.6	1.38
s400	753	896	84.0%	2,446	47.0	20.8	0.6	0.26	896	64.9	27.6	0.9	1.05
s420	738	738	100.0%	4,348	117.7	54.2	1.0	0.23	738	158.8	71.6	1.3	1.72
s444	813	1,070	76.0%	3,369	49.8	21.7	0.9	0.25	1,070	69.7	29.2	1.2	1.13
s510	738	738	100.0%	3,224	79.1	32.4	1.3	0.39	738	148.2	58.3	1.3	1.79
s526	720	820	87.8%	2,241	44.9	19.3	0.8	0.36	820	71.9	29.0	1.0	1.26
s641	2,231	3,444	64.8%	14,421	270.7	115.3	5.7	0.39	3,444	325.4	135.7	13.4	3.90
s713	4,922	43,624	11.3%	32,124	310.8	125.5	17.2	0.53	43,624	391.7	152.2	234.3	5.37
s820	984	984	100.0%	4,570	84.8	33.2	2.3	0.49	984	143.8	54.8	2.4	2.44
s832	996	1,012	98.4%	4,705	85.8	33.1	2.4	0.50	1,012	147.4	55.1	2.5	2.50
s838	2,018	2,018	100.0%	13,374	197.0	93.2	5.3	0.39	2,018	346.3	160.0	7.2	3.54
s953	2,312	2,312	100.0%	9,876	102.3	44.5	4.0	0.41	2,312	178.1	73.5	5.1	2.22
s1238	3,684	7,118	51.8%	19,262	248.7	94.0	19.5	1.01	7,118	487.8	177.0	41.5	5.83
s1423	45,198	89,452	50.5%	174,188	395.0	165.7	74.0	0.42	89,452	450.2	186.5	553.6	6.19
s1488	1,916	1,924	99.6%	15,043	119.6	46.6	6.5	0.43	1,924	187.3	71.1	6.9	3.60
s1494	1,927	1,952	98.7%	15,194	119.7	46.4	6.6	0.43	1,952	187.6	70.9	7.0	3.60
s5378	21,890	27,046	80.9%	129,996	415.6	181.3	98.7	0.76	27,046	472.8	206.6	214.9	7.95
s9234.1	59,854	489,708	12.2%	921,161	713.5	309.7	1,856.8	2.02	489,708	1,348.4	584.9	15,632.0	31.92
s13207.1	476,145	2,690,738	17.7%	6,035,082	3,186.6	1,249.1	14,560.0	2.41	-	-	-	>30,000	-
s35932	58,657	394,282	14.9%	1,181,808	120.4	42.7	6,920.3	5.86	394,282	251.9	93.4	24,964.0	63.32
s38417	1,138,194	2,783,158	40.9%	9,759,983	1043.9	449.6	29,225.0	4.42	-	-	-	>30,000	-
s38584.1	334,927	2,161,446	15.5%	3,500,558	1,608.4	699.3	15,455.0	4.42	-	-	-	>30,000	-

TABLE II: Generation of non-robust tests for the combinational portion of ISCAS 89 circuits

$$f(P_1) = (x_1)(\overline{x_1} + \overline{x_4})(\overline{x_2} + \overline{x_4})(x_1 + x_2 + x_4)(\overline{x_4})(\overline{x_2})$$
$$(x_7)(x_4 + x_7)(x_5 + x_7)(\overline{x_4} + \overline{x_5} + \overline{x_7})(x_5)$$

This function can easily be shown to be unsatisfiable. Hence we can conclude that all paths that extend from P_1, namely (P_1, x_{11}) and (P_1, x_{12}), are unsensitizable and can be removed from further consideration. The next set of paths considered by the incremental search algorithm would now extend P_p through x_8. ■

It is worth noting that there are other approaches to identify non-robust untestable faults such as [6] [10] which use logic implication to identify untestable faults. Even though they are substantially faster than the methods which target paths, they are not complete in that they cannot guarantee 100% identification of untestable faults. A similar method is used in [8] to identify robust dependent and functionally unsensitizable faults.

IV. Experimental Results

The prototype of the presented algorithm is implemented in C++ and integrated with the SAT solver GRASP [13]. It was run on a workstation with Intel Pentium II 300 MHz CPU and 256 MB of memory running Linux. The result of the experiment is shown in Table II.

The first four columns present general information about the benchmark circuits. In the remaining columns, the number of SAT calls and average problem size in terms of number of variables and clauses for both incremental method and the method in [2] are presented. Average execution time per each SAT call as well as total execution time is also reported.

We implemented the algorithm used in [2] for comparison with our method within our framework; for each path in a circuit, we identify a corresponding formula, add constraints clauses and solve the SAT problem. Note that in the actual implementation, the CNF formula for the entire circuit is generated only once.

From the experimental results, we can observe that the savings we gain by using incremental satisfiability accrue for circuits with a large number of untestable faults such as s713, s1238, s1423, s9234.1 and below.

Note that our method can generate complete test sets for three circuits (s13207.1, s38417, s38584.1) which we could not finish within the given time limit (30,000 seconds) using the explicit path enumeration method.

It is also worthwhile to note that even though our method requires more SAT calls in general, the time per each SAT call (reported in milliseconds in the table) of our method is significantly smaller than that of the explicit method because

of the application of ISAT which uses the decision sequence from previous problems.

Also note that average problem size in terms of number of clauses and variables of incremental method is substantially smaller than that of explicit method. Although the size of a SAT problem is not necessarily a measure of the difficulty of the problem, it is generally considered that larger SAT problems are harder to solve than smaller problems.

V. Conclusions

In this paper, we presented a method of solving incremental satisfiability problems (ISAT) which can check the satisfiability of a family of related functions. The application of ISAT to delay fault testing is also presented. Promising results were obtained when we applied this method to the generation of non-robust tests for the combinational part of ISCAS 89 benchmark circuits.

Incremental satisfiability can be applied to the problems in other EDA domains, such as timing analysis and logic verification. We are currently working on applying this method to timing analysis of sequential circuits.

Also, the application of ISAT is not limited to solving CNF based satisfiability problems but can be applied to problems where there are general constraints on Boolean variables that are not expressed in CNF. We are presently implementing the extension of ISAT to general constraints.

References

[1] R. Bayardo Jr. and R. Schrag, “Using CSP Look-Back Techniques to Solve Real-World SAT Instances,” National Conference on Artificial Intelligence, pp. 203-208, 1997.

[2] C. -A. Chen and S. K. Gupta, “A Satisfiability-Based Test Generator for Path Delay Faults in Combinational Circuits,” Design Automation Conference, pp. 209-214, 1996.

[3] K. -T. Cheng and H. -C. Chen, “Delay Testing For Non-Robust Untestable Circuits,” International Test Conference, pp. 954-961, 1993.

[4] K. -T. Cheng and H. -C. Chen, “Generation of High Quality Non-Robust Tests for Path Delay Faults,” Design Automation Conference, pp. 365-369, 1994.

[5] K. Fuchs, F. Fink and M. H. Schulz, “DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults,” Trans. on CAD, vol. 10, pp. 1323-1335, Oct. 1991.

[6] K. Heragu, J. H. Patel and V. D. Agrawal, “Fast Identification of Untestable Delay Faults Using Implication,” International Conference on Computer Aided Design, pp. 642-647, 1997.

[7] J. N. Hooker, “Solving the Incremental Satisfiability Problem,” Journal of Logic Programming, vol. 15, pp. 177-186, 1993.

[8] S. Kajihara, K. Kinoshita, I. Pomeranz and S. M. Reddy, “A Method for Identifying Robust Dependent and Functionally Unsensitizable Paths,” International Conference on VLSI Design, pp. 82-87, 1997.

[9] J. Kim, J. P. M. Silva, H. Savoj and K. A. Sakallah, “RID-GRASP: Redundancy Identification and Removal Using GRASP,” International Workshop on Logic Synthesis, 1997.

[10] Z. Li, Y. Min and R. K. Brayton, “Efficient Identification of Non-Robustly Untestable Path Delay Faults,” International Test Conference, pp. 992-997, 1997.

[11] C. J. Lin and S. M. Reddy, “On Delay Fault Testing in Logic Circuits,” Trans. on CAD, vol. 6, pp. 694-703, 1987.

[12] A. Pierzynska and S. Pilarski, “Non-Robust Versus Robust,” International Test Conference, pp. 123-131, 1995.

[13] J. P. M. Silva and K. A. Sakallah, “GRASP- A New Algorithm for Satisfiability,” International Conference on Computer Aided Design, pp. 220-227, 1996.

[14] H. Zhang, “SATO: An Efficient Propositional Prover,” International Conference on Automatic Deduction, pp. 272-275, 1997.