Incremental ATPG Methods for Multiple Faults under Multiple Fault Models

Masahiro Fujita Naoki Taguchi Kentaro Iwata
University of Tokyo

Alan Mishchenko
University of California, Berkeley

***Abstract*—We propose a general framework for incremental automatic test pattern generation of combinational circuits targeting multiple faults under multiple fault models. Not only standard fault models, such as stuck-at faults, but also any functional fault models can be targeted, once the fault model is given in terms of the resulting logic functions under the faults. Given sufficient time, the proposed methods can automatically generate complete sets of test patterns for multiple standard and/or custom faults. Although there are exponentially many combinations of multiple faults, the proposed SAT based formulation can generate test patterns for all of them incrementally. As a consequence, as long as the required numbers of test patterns are not so many, which has been the case for all of our experimental results, we can finish generating complete sets of test patterns. We have implemented the proposed methods, and through experiments we show that complete sets of test patterns targeting multiple faults under various fault models for circuits having up to tens of thousand of gates can be automatically generated. As any functional faults can be dealt with the proposed methods, they can also be applied to circuit transformation based logic synthesis.**

I. Introduction

As the semiconductor technology continues to shrink, we have to expect more and more varieties of variations in the process of manufacturing especially for large chips. This may result in situations where multiple faults as well as non-traditional faults are actually happening in the chips. This suggests that single-fault models may not well capture defects in deeply shrunken devices. Also, we may need more complicated fault models than traditional stuck-at faults, such as functional faults.

ATPG for multiple faults, however, has been considered to be very expensive and except for very small circuits, it is practically impossible as there as so many fault combinations for multiple faults. For example, if there are m possibly faulty locations in the circuit, there are $3^m - 1$ fault combinations for multiple stuck-at faults. If m is 10,000, there are $3^{10,000} - 1 \approx 10^{4771}$ fault combinations. If we check whether each fault can be detected by the current set of test patterns, that process will take almost forever.

Traditionally ATPG processes include pattern generation as well as fault simulation in order to eliminate detectable faults with the current sets of test patterns from the sets of target faults. The problem here is the fact that fault simulators represent all faults explicitly. Therefore, fault simulators do not work if the numbers of fault combinations becomes exponentially large which is the case if we target all of multiple fault combinations.

We resolve this problem by representing fault lists "implicitly" and combine the detectable fault elimination process with test pattern generation process as incremental SAT (Satisfiability checking) problems.

Moreover, in order to deal with varieties of fault models, functional modeling methods for faults in general are introduced. That is, various faults for each gate in the circuit are represented as resulting faulty logic functions. When fault happens in the gate, such logic functions show which kind of functionality can be found at the inputs and output of the faulty gate. This is realized with what we call "parameter" variables. Basically if values of the parameter variables are all zero, there is no fault in the gate. If some or all variables are non-zero, however, there are corresponding faulty functions defined by the logic functions with parameter variables.

For example, if we like to model stuck-at 0 fault at the output of a gate and stuck-at 1 faults at the inputs of a gate, we introduce the logic circuit (or logic function) with parameter variables, p, q, r as shown in Figure 1. Here the target gate is an AND gate and its output is replaced with the circuit shown in the figure. That is, the output of the AND gate, c is replaced with $((a \vee p) \wedge (b \vee q)) \wedge \neg r$, which becomes 0 (corresponding to stuck-at 1 on signal c) when $r = 1$, b (corresponding to stuck-at 1 fault on signal a) when $p = 1, q = r = 0$, and a (corresponding to stuck-at 1 fault on signal b) when $q = 1, p = r = 0$. When a stuck-at 1 fault happens on the input signal a, that value becomes 1 and so the logic function observed at the output, c is b assuming that there is no more fault in this gate. If all of p, q, r are 0, the behavior remains the same as original AND function which is non-faulty.

Stuck-at faulty behaviors for each location are realized with these additional circuits. That is, circuits with additional ones can simulate the stuck-at 1 and 0 effects by appropriately setting the values of $p, q, r, ...$. For m possibly faulty locations, we use m of $p, q, r, ...$ variables. As we deal with multiple faults, these circuits for modeling various faults should be inserted into each gate in the circuit. By introducing appropriate circuit (or logic functions) with parameter variables, varieties of multiple faults can be formulated in a uniform way.

As related works with the proposed methods, the ATPG problems can be naturally formulated as QBF (Quantified Boolean Formula). We are solving these through repeated application of SAT solvers, which was first discussed under FPGA synthesis in [2] and in program synthesis in [3]. [4] discusses the general framework on how to deal with QBF only with SAT solvers.

We have implemented the proposed ATPG methods and a

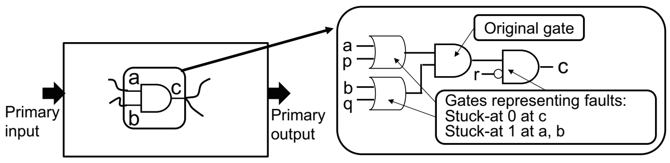

Fig. 1. Modeling stuck-at faults at the inputs and the output of a gate

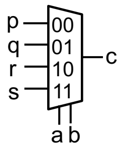

Fig. 2. Modeling functional faults

framework based on them on top of ABC tool [1]. Experimental results show that we can perform ATPG for all combinations of various multiple faults for all ISCAS89 circuits. As shown in the experiments, complete sets of test vectors (exclusive of redundant faults) are successfully generated for several fault models for all ISCAS89 circuits.

The fault models defined through their resulting logic functions can be considered to be transformations of gates. That is, under faults, each gate change its behavior, which are represented by the logic functions and correspond to circuit transformations. By interpreting this way, the proposed methods can also be used for logic synthesis based on circuit transformations. If there are p transformations possible on a gate and there are in total m gates in the circuit, logic synthesis based on the proposed method can search synthesized circuits out of p^m possible multiple transformations. Although this is a very interesting topic, we reserve it for a future research topic.

The rest of the paper is organized as follows. In the next section we introduce a method by which various fault models can be defined. Then the following section presents the proposed test pattern generation methods based on incremental SAT and implicit representations of fault lists. The section which shows experimental results follows. The final section gives concluding remarks including a brief discussion on relationships with the previous works.

II. Representing various fault models as logic functions

As discussed in the previous section, we introduce a mechanism by which various fault models can be represented as logic formulae or functions. In the following, for easiness of explanation and also for easiness of implementation as a tool, we assume all gates in the circuit are two-input AND gates with possibly inverters as their inputs and outputs. This is exactly AIG (AND-INVERTER-GRAPH) used in the tool, ABC [1]. As generalization for general circuits having varieties of gates is straightforward, in this paper, we assume the target circuit is represented as AIG.

In general, for each gate, we describe how its logic function at its output with respect to its inputs changes under a given fault model. For example, if the stuck-at 0 and stuck-at 1 faults at the output of an AND gate are the only possible faults (no faults assumed in its inputs), the logic formula or function we define becomes $(a \wedge b) \wedge \neg p) \vee q$. Here if both of p and q are 0, there is no fault. But if not there is either stuck-at 1 or stuck-at 0 fault at the outout of the AND gate.

As we use the same fault models for all gates in the circuit, we use $p, q, r, ...$ as fault parameter variables whereas a and b are reserved as the inputs of the gates. c is reserved for the output of the gate. Precisely speaking, all of those variables have indexes which identify the gates. If there are m gates in the circuit, that index can range 1 to m. Please remember that in this paper, we assume all of the gates in the circuit are AND gates for easiness of explanation and implementation. This logic function shows the values realized at the output of the gate with and without faults. When all of $p, q, r, ...$ are zero, The gate behaves correctly without any fault. If some or all of $p, q, r, ...$ are non zero, the faulty value at the output of the gate is given by the logic function.

It is clear that by changing the formula, we can define various fault models. Here are some logic functions for varieties of fault models. For all the examples below, we assume non-faulty correct function is two-input AND operation. Also, $F?G : H$ means if F then G else H.

1) If faulty, output becomes constant 0: $p?\neg p : a \wedge b$.
2) If faulty, output becomes constant 1: $p?p : a \wedge b$.
3) If faulty, output becomes OR: $p?(a \vee b) : a \wedge b$.
4) If faulty, output becomes NOR: $p?(\neg a \wedge \neg b) : a \wedge b$.
5) If faulty, output becomes NAND: $p?(\neg a \vee \neg b) : a \wedge b$.
6) When $p = 1$, stuck at 0 fault on the input signal, a. When $q = 1$, stuck-at 0 fault on the input b. Also when $r = 1$, stuck-at 1 fault at the output, c. Multiple faults may happen within each gate: $(a \wedge \neg p) \wedge (b \wedge \neg q)) \vee r$.
7) One four-input multiplexer is introduced. Its output is connected to c, and its four inputs are connected to p, q, r, s. The multiplexer is controlled by the values of a and b: $a?(b?p : q) : (b?r : s)$. In circuit, this is equivalent to the one in Figure 2.

The last one is a special one in the sense that the non-faulty behavior is not given when all of p, q, r, s are zero. As all two-input logic functions can be defined by setting appropriate values to p, q, r, s, one of them is the correct one and all the others are faulty ones. If the original gate function is AND, the correct behavior is given when $p = 1, q = r = s = 0$. So the test pattern process needs to take care of this, but all the others remains the same as previous examples. Because the fault can realize any logic function with two-inputs, its test generation is very hard if multiple faults are allowed.

III. Proposed test pattern generation method with incremental SAT formulation

In this section, we present a test pattern generation method which, given sufficient time, can generate complete sets of test vectors targeting all combinations of multiple faults under multiple fault models in given combinational circuits. The target fault models can be specified by the parameter circuits shown above. The test pattern generation problem can be formulated as incremental SAT (Satisfiability checking). That is, the SAT problem, $(\mathit{faulty\ circuit\ output}) \neq (\mathit{correct\ circuit\ output})$, is satisfiable, the solution for the primary inputs is a test pattern. If this is UNSAT (unsatisfiable), that faults are redundant.

Now we define a formula for the test pattern generation using the fault models with parameter variables shown in the previous section. Let In be the set of primary inputs of given combinational circuits. Also, let X be the set of parameter variables. (p_i, q_i, r_i, ...) for modeling faults. As the parameter variables, $p, q, r, ...$, introduced in the previous section may have different values for each gate of the circuit, they have index, i for each gate. Let $Correct(In)$ is the output function of the circuit without faults, e.g., the output value is $Correct(in_1)$ for the input value in_1. In the case of faulty circuits, let $Faulty(In, X)$ be the output function under the multiple faults defined by X. For example, under the fault of $X = x_1$, the output is $Faulty(in_i, x_1)$ when $In = in_1$. For easiness of explanation, we assume there is only one output in the circuits. Generalization of multiple outputs is simple and straightforward.

Then in order to generate a test pattern for a fault the following SAT problem should be solved:

$$\exists In, X. Faulty(In, X) \neq Correct(In) \qquad ...(1)$$

Please note that this is a normal SAT problem and says some fault can be detected by some input pattern, as under that input pattern the non-faulty and faulty circuits behave differently. Let the solution values of variables be $In = in_1, X = x_1$. Now we have found that the fault corresponding to x_1 can be detected by the input, in_1.

We put together elimination of detectable faults and test pattern processes into a single SAT problem. In order to eliminate the faults that can be detected by the test pattern, in_1, we add the following constraint to (1):

$$Fauty(in_1, X) = Correct(in_1)$$

This constrains X to be the ones which behave correctly with the test pattern, in_1, that is, undetectable under in_1.

So the next step of our ATPG process is to solve the following SAT problem:

$$\begin{gathered}\exists In, X.(Faulty(In, X) \neq Correct(In))\\ \wedge(Faulty(in_1, X) = Correct(in_1)) \qquad ...(2)\end{gathered}$$

where in_1 is the solution of (1) above.

Let the solution values of variables, (In, X), for (2) be (in_2, x_2) respectively. Then in_2 becomes the second test pattern. It detects some fault which cannot be detected by the test pattern, in_1.

We keep doing this until there is no more solution. Here we assume the following SAT problem has a solution.

$$\begin{gathered}\exists In, X.(Faulty(IN, X) \neq Correct(In))\\ \wedge(Faulty(in_1, X) = Correct(in_1))\\ \wedge(Faulty(in_2, X) = Correct(in_2)) \wedge ...\\ \wedge(Faulty(in_{n-1}, X) = Correct(in_{n-1})) \ ...(3)\end{gathered}$$

And the following SAT problem has no solution, that is unsatisfiable.

$$\begin{gathered}\exists In, X.(Faulty(In, X) \neq Correct(In))\\ \wedge(Faulty(in_1, X) = Correct(in_1))\\ \wedge(Fauty(in_2, X) = Correct(in_2)) \wedge ...\\ \wedge(Faulty(in_{n-1}, X) = Correct(in_{n-1}))\\ \wedge(Faulty(in_n, X) = Correct(in_n)) \ ...(4)\end{gathered}$$

As (3) has a solution and (4) does not have a solution, the test patterns, $in_1, in_2, ..., in_n$ can detect all of the detectable faults, as the unsatisfiability of the formula (4) guarantees that there is no more detectable fault. Please note that redundant faults are automatically excluded from the target faults in the above SAT formulation.

The numbers of test vectors required to detect all faults, or in other words, the performance of the ATPG algorithm depends on how many times the formula (3) becomes satisfiable, i.e., numbers of iterations of SAT solving.

As can be clearly seen from the above, the SAT problems to be solved are pure "incremental SAT" problems. Here "pure" means that the formulae are updated to have more constraints, that is, a following formula is a super set of the previous formulae. Constraints are never deleted. Therefore, all learning and conflicts found so far are guaranteed to be all valid in the following formulae, and reasoning in the previous formula can simply be continued in the following formula. The set of the SAT instances (or formulae) can be considered as a single SAT problem, which should finally be unsatisfiable. So the overall process of the proposed ATPG method is just to solve single SAT problem to make sure it is unsatisfiable, allowing dynamic addition of more constraints during the SAT reasoning process.

IV. Experimental results

We have implemented the proposed incremental ATPG framework on top of ABC tool [1]. In order to demonstrate how much the proposed ATPG method works, all ISCAS89 circuits are processed to generate complete test patterns for all combinations of multiple faults under various fault models. For easiness of experiments, given ISCAS89 circuits are first transformed into AIG representation, and user-defined fault models are inserted into each AND gate.

The characteristics of ISCAS89 circuits in AIG format is shown in Table I. Name is the name of an ISCAS89 benchmark, and PI/PO/FF/AND are the numbers of primary inputs, outputs, flipflops, and AIG nodes used to represent the circuits. As the column, AND, shows the numbers of AND nodes in the circuits, these correspond to the numbers of possibly faulty locations which incorporate the fault models defined by user-specified logic functions.

The ATPG results are shown in Table II and III. As small circuits take less than a second, only larger circuits of ISCAS89 are shown. We show here the results of six different fault models (1) to 6)) in section II. Tests are the numbers of test vectors generated. Time is the processing time on a desktop computer having Linux kernel 2.6.32 64-bit, Dual Xeon E5-2690 2.9GHz, and 128GB memory. We have successfully generated complete sets of test patterns for all ISCAS89 circuits for all of the fault models, which we believe is the first time ever for multiple faults. The fault models and implicit representations of faults work well even for circuits having more than ten thousands of gates. Please note that these

Name	PI	PO	FF	AND
s820	18	19	5	345
s832	18	19	5	356
s838.1	34	1	32	336
s953	16	23	29	347
s1196	14	14	18	477
s1238	14	14	18	532
s1423	17	5	74	462
s1488	8	19	6	663
s1494	8	19	6	673
s5378	35	49	179	1389
s9234	19	22	228	1958
s13207	31	121	669	2719
s15850	14	87	597	3560
s35932	35	320	1728	11948
s38417	28	106	1636	9219
s38584	12	278	1452	12400

TABLE I
Characteristics of ISCAS89 circuits in AIG[1]

Name	Constant 0 at fault: p?(~p):(a&b)		Constant 1 at fault: p?(p):(a&b)		OR at fault: p?(a+b):(a&b)	
	Tests	Time (s)	Tests	Time (s)	Tests	Time (s)
s820	70	0.05	98	0.32	192	2.23
s832	61	0.04	89	0.24	304	5.29
s838.1	81	0.07	189	1.07	353	28.92
s953	69	0.06	117	0.98	184	4.29
s1196	127	0.2	145	0.72	280	6.27
s1238	123	0.18	140	0.75	229	5.11
s1423	98	0.21	73	0.22	134	1.44
s1488	119	0.22	104	0.47	244	7.59
s1494	108	0.18	135	0.6	287	11.76
s5378	285	4.56	231	8.79	311	89.9
s9234	463	22.91	271	22.66	523	952.52
s13207	713	147.12	350	85.4	529	940.1
s15850	634	125.95	326	64.65	489	4407.7
s35932	1048	5332.44	718	2333.59	164	9417.82
s38417	1174	1600.62	540	1073.7	1013	21552.74
s38584	2309	6860.81	633	1174.87	711	4165.27

TABLE II
Complete test generation of all multiple user-defined faults for ISCAS89 circuits (1)

sets of test vectors can completely test all combinations of the multiple faults as long as they are not redundant.

As can be seen from the tables, for the largest three benchmark circuits, S35932, S38417, and S38584, which have around or more than ten thousands of AND gates, it takes much more time than the others. The numbers of test patters for complete multiple fault testing, however, still remain around one or two thousands, which make the proposed ATPG methods applicable to these sizes of circuits.

There are cases where for some fault models and circuits, it takes a very long time compared with others, such as S838.1 and S9234 under the fault model which changes AND into NOR. We are investigating why these are happening now. Also, the faults models which change AND into OR,

Name	NOR at fault: p?(~a&~b):(a&b)		NAND at fault: p?(~a+~b):(a&b)		Faults for 6): ((a&~p)&(b&~q))+r	
	Tests	Time (s)	Tests	Time (s)	Tests	Time (s)
s820	348	5.35	337	2.71	105	2.12
s832	334	4.19	461	5.71	126	3.86
s838.1	850	97206.33	989	154.81	224	9.97
s953	92	1.01	164	1.31	92	2.58
s1196	249	3.83	260	1.75	162	5.51
s1238	350	5.9	459	8.07	173	8.03
s1423	164	1.18	192	2.06	126	1.61
s1488	338	12.65	294	6.67	169	6.29
s1494	317	8.57	336	7.66	156	6.33
s5378	303	480.98	355	83.92	309	56.23
s9234	531	73097.91	562	655.44	438	188
s13207	668	399.24	487	270.79	537	418.22
s15850	577	666.49	609	483.32	554	499.34
s35932	492	5031.7	242	19814.98	1088	13043.61
s38417	1096	10373.17	1016	18624.9	930	7957.68
s38584	1086	7736.2	1132	7293.37	1517	12479.45

TABLE III
Complete test generation of all multiple user-defined faults for ISCAS89 circuits (2)

NOR, and NAND under faults are conceptually very close, but their ATPG performances are sometime significantly different. These indicate that there are lots of rooms to be improved in the proposed methods.

V. Concluding remarks

We have shown an ATPG framework for multiple various faults with user-define fault models and implicit representation of fault lists. The problems can be naturally formulated as QBF (Quantified Boolean Formula), but solved through repeated application of SAT solvers. There have been works in this direction [2], [3], [4]. From the discussion in this paper we may say that those problems could also be processed as incremental SAT problems instead of QBF problem, and those incremental SAT problems can essentially be solved as single (unsatisfiable) SAT problems allowing additional constraints in the fly, which could be much more efficient.

The experimental results shown in this paper are very preliminary. Although complete sets of test patterns for various multiple faults have been successfully generated, which is, as long as we know, the first time ever, there are lots of rooms in the proposed methods to be improved and to be extended. One of them is to try to compact the test pattern sets.

References

[1] Robert K. Brayton, Alan Mishchenko: ABC: An Academic Industrial-Strength Verification Tool, *22nd International Conference on Computer Aided Verification (CAV 2010)*, pp.24–40, 2010.

[2] Andrew Ling, P. Singh, and Stephen D. Brown: FPGA Logic Synthesis Using Quantified Boolean Satisfiability, *SAT*, 2005.

[3] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, Vijay A. Saraswat: Combinatorial sketching for finite programs, *ASPLOS 2006*, pp.404-415, 2006.

[4] Mikolas Janota, William Klieber, Joao Marques-Silva, Edmund M. Clarke: Solving QBF with Counterexample Guided Refinement, *SAT 2012*, pp.114-128, 2012.