IPSJ Transactions on System LSI Design Methodology Vol.10 39–44 (Feb. 2017)

[DOI: 10.2197/ipsjtsldm.10.39]

Regular Paper

An Accelerating Technique for SAT-based ATPG

Yusuke Matsunaga[1,a)]

Received: June 2, 2016, Revised: September 7, 2016,
Accepted: October 19, 2016

Abstract: This paper describes an accelerating technique for SAT based ATPG (automatic test pattern generation). The main idea of the proposed algorithm is representing more than one test generation problems as one CNF formula with introducing control variables, which reduces CNF generation time. Furthermore, learnt clauses of previously solved problems are effectively shared for other problems solving, so that the SAT solving time is also reduced. Experimental results show that the proposed algorithm runs more than 3 times faster than the original SAT-based ATPG algorithm.

Keywords: ATPG, SAT, CNF

1. Introduction

The idea of formalizing test generation problem as CNF-SAT problem and using SAT solver was proposed by Larrabee a couple of decades ago [1]. At that time, however, the algorithm for SAT solving was not mature enough, and thus SAT based ATPG techniques were not competitive against the existing structural ATPG techniques. Currently, the success of development of efficient SAT solvers such as Chaff [2], [3], GRASP [4] and MINISAT [5] enables SAT based ATPG more practical. Besides the performance, SAT based ATPG has an advantage that it is easy to treat several constraints at once if they are represented in a CNF formula.

This paper proposes an accelerating technique for SAT based ATPG. Because nowadays SAT solvers are efficient enough, there seems no room to improve SAT solving time. So our target is not SAT solving time itself, but total CPU time. The main idea is to share a CNF-formula for more than one faults with introducing control variables, which leads the reduction of CNF generating time. Also this technique has side effect that learnt clauses of previously solved problems are utilized for other remaining problems, which fits to nowadays efficient SAT algorithm very well. As a results, the proposed algorithm significantly reduce the total CPU time. The effects are obvious especially for large designs.

The rest of the paper is organized as follows: Section 2 reviews the existing techniques for SAT based ATPG. In Section 3, proposed techniques are described, and the experimental results are shown in Section 4. Finally, Section 5 concludes the paper.

2. SAT Based ATPG

This section describes basic definitions and the existing techniques for SAT based ATPG.

2.1 Basic Definitions

This paper treats combinational circuits. We assume that synchronous sequential circuit can be transformed into combinational circuit with replacing FFs (flip-flops) to scan-type FFs.

A combinational circuit can be viewed as a **graph** where each gate corresponds to a **vertex** and each connection between gates corresponds to an **edge**. When there is an edge (u, v) from vertex u to vertex v, i.e. there is a connection from gate u to gate v, u is called as **fan-in** of v. The set of all the fan-ins of v is denoted as $FI(v)$. Also, v is called as **fan-out** of u. The set of all the fan-outs of u is denoted as $FO(u)$

A sequence of vertices such that there are edges corresponding to all the adjacent pairs of the sequence is called a **path**. That is, if $(v_0, v_1, v_2, v_3, ..., v_{n-1}, v_n)$ is a path, $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ must be edges. The first vertex of a path is called the **start point**, and the last vertex is called the **end point**. When there is a path whose start point is u and whose end point is v, u is called **transitive fan-in** of v. The set of all the transitive fan-ins of v is denoted as $TFI(v)$. Also, v is called **transitive fan-out** of u. The set of all the transitive fan-outs of u is denoted as $TFO(u)$.

When, any paths from v to the primary outputs contain a vertex s, s is called as a **dominator** of v. The set of all the dominators of v is denoted as $DOM(v)$. It is obvious from the definition that every dominators of v appear in any path from v to the primary outputs. Combinational circuit does not have cycles, thus each pair $(s, t) \in DOM(v)$ satisfies $s \in DOM(t)$ or $t \in DOM(s)$, exclusively. A vertex u such that $u \in DOM(v)$ and $\forall w \in (DOM(v) \setminus u), u \notin DOM(w)$ is called the **immediate dominator** of v, and is denoted as $IDOM(v)$. There is at most one immediate dominator for every vertex.

A **Maximal fan-out free cone** (**MFFC** in short) is a connected subgraph induced by a set of vertices having the same $R_{MFFC}(v)$ value in the following Eq. (1).

$$R_{MFFC}(v) = \begin{cases} v & IDOM(v) = \phi \\ R_{MFFC}(IDOM(v)) & \text{otherwise} \end{cases} \quad (1)$$

1 Kyushu University, Fukuoka 819–0395, Japan
a) yusuke_matsunaga@ieee.org

MFFC divides a combinational circuit with no overlap. For each MFFC, there exists only one vertex v such that $R_{MFFC}(v) = v$, v is called the **root** of the MFFC.

A vertex having more than one fan-outs is called a **fanout vertex**. Cutting off the entire circuit at each fanout vertex splits the circuit into a set of subcircuits. Such a subcircuit is called A **fanout free region** (**FFR** in short), and a fan-out vertex is called the **root** of the FFR. It is clear that vertices in an FFR except the root have only one fan-out. It is also clear that an FFR is always contained in an MFFC (, or an FFR itself becomes an MFFC).

2.2 Test Pattern Generation Problem

Consider a fault f. Generally, the behavior of the circuit with a fault f (we call such a circuit as the **faulty circuit of** f or the **faulty circuit** when f is obvious) is different from the behavior of the circuit without any faults (we call such a circuit as **good circuit**). The test pattern generation problem for a fault f is a problem finding an input pattern which differentiates the outputs of the faulty circuit of f and good circuit. Such an input pattern is called a **test pattern** or a **test vector**, and we say a test pattern **detects** the fault f. There may be a case when the behavior of the faulty circuit and the behavior of the good circuit are the same. In that case, we can not find test patterns of the fault. Actually, such a fault does not effect the circuit behavior, and is called as a **redundant fault**.

In this paper, the single stuck-at fault model is considered, which means there is only one fault at once, and faulty behavior is sticking the value of an input/output to 0 or 1. A fault which sticks the value to 0 is called a **stuck-at-0** fault and a fault which sticks the value to 1 is called a **stuck-at-1** fault. The total number of faults to be considered is $E \times 2$, where E denotes the number of edges (connections) of the circuit. Note that there are faults whose behaviors are the same. When the faulty circuit of a fault f and the faulty circuit of a fault g are functionally equivalent, a test pattern for f can detect g and visa versa. For example, a stuck-at-0 fault of an input of an AND gate and a stuck-at-0 fault of the output of the gate are equivalent. As the name 'equivalent' implies, this relation is equivalent relation, and faults are divided into equivalent classes. So, we can consider one fault for each equivalent class. Such a fault is called the representative fault. However, finding exact equivalent classes for all the faults are hard problem, so heuristics finding subset of equivalent classes using structural information are used in the real application.

2.3 SAT Encoding for Test Pattern Generation

The basic approach to solve test pattern generation problem using SAT solver [1] is described below. At first, we compose a virtual circuit as shown in **Fig. 1**, which consists of the good circuit and the faulty circuit of a fault f. The primary inputs of the composed circuit connect to the corresponding primary inputs of the good circuit and the faulty circuit. There are XOR gates calculating the difference of the corresponding outputs of the good circuit and the faulty circuit. Finally, there is an OR gate whose fan-ins are XOR gates, and the output of the OR gate is the primary output of the composed circuit.

If any of the corresponding outputs of the good circuit and the faulty circuit differ, the composed circuit outputs '1'. So, the test pattern generation problem can be transformed to the problem finding an input pattern which makes the output of the composed circuit '1', which can be solved using SAT solver if we can encode the circuit structure into a CNF-formula.

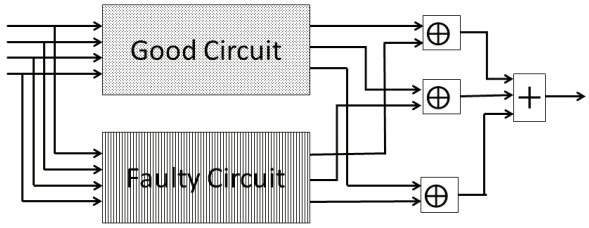

Fig. 1 The composed circuit for test pattern generation.

The encoding is rather straightforward.

(1) Assign Boolean variables for every edges (connections).

(2) For each gate, generate a CNF formula representing the relation between the inputs and the output of the gate.

(3) Conjoin all the CNF formula.

There are several methods to generate a CNF formula representing the relation between the inputs and the output. A simple method is as follows. Consider 3-AND gate with a, b, and c as inputs and x as an output. Implications among them are like this:

- If any of a, b, or c is 0, x is 0.
- If a, b, and c are all 1, x is 1.
- If x is 1, a, b and c are all 1.

Let α and β be formulas. A formula representing "α implies β" is $\neg\alpha \vee \beta$. The above implications are transformed into the next formulas:

- $a \vee \neg x$, $b \vee \neg x$, $c \vee \neg x$
- $\neg a \vee \neg b \vee \neg c \vee x$
- $\neg x \vee a$, $\neg x \vee b$, $\neg x \vee c$

Actually, the first one and the last one are the same, because the original implications are contraposition relation. Like this way, from the input combinations which makes the output '0' and '1', we can easily generate a CNF-formula for any types of logic gate. The size of the CNF formula is $O(N)$, where N is the size of the circuit.

There are couple of existing techniques for efficiency. First one is sharing with the good circuit and the faulty circuit. Consider a fault is in gate v, the effect of the fault can be propagated only through the transitive fan-outs of v, that means the values of other gates of the faulty circuit are equal to the corresponding the gates of the good circuit. So, only gates in $TFO(v)$ are necessary. The values of other gates are taken from the corresponding gates of the good circuit. Second one is reduction of the entire circuit. As described above, the fault effect is propagated only through the gates in $TFO(v)$. So, gates which do not affect the values of the gates in $TFO(v)$ are also not necessary even for the good circuit. That is, only gates in $TFI(TFO(v))$ are necessary to be considered.

Larrabee proposed another technique to speed up SAT based ATPG, which is called 'D-chain' constraint [1]. D-chain is a constraint that guarantees the existence of fault propagation paths.

For that purpose, a new variable for each gate is introduced. Let D_x be the D-chain variable for gate x, and G_x and F_x are the variables representing the output values of the good circuit and the faulty circuit of x, respectively. A necessary condition that gate x can propagate the fault effect to any of the primary outputs is that the value of the good circuit and the value of the faulty circuit are different. This condition is represented in the following formula.

$$(\neg D_x \vee G_x \vee F_x)(\neg D_x \vee \neg G_x \vee \neg F_x) \quad (2)$$

Suppose y and z are the fanouts of x. Another necessary condition is that y or z can also propagate the fault effect to any of the primary outputs, which is represented in the following formula.

$$(\neg D_x \vee D_y \vee D_z) \quad (3)$$

2.4 Heuristics in Modern SAT Algorithms

Modern SAT solvers such as Chaff [2], [3], GRASP [4] and MINISAT [5] employs a couple of heuristics for efficient searching. Some of them which are related to the proposed method are described here.

One is called 'conflict driven learning'. Suppose the following set of formulas φ

$$C_1 = \neg x_1 \vee x_2$$
$$C_2 = \neg x_1 \vee x_3 \vee x_5$$
$$C_3 = \neg x_2 \vee x_4$$
$$C_4 = \neg x_3 \vee \neg x_4$$

With assigning $x_5 \leftarrow 0$ and $x_1 \leftarrow 1$, we have the following implications.

$$x_1 = 1 \xrightarrow{C_1} x_2 = 1$$
$$x_1 = 1 \wedge x_5 = 0 \xrightarrow{C_2} x_3 = 1$$
$$x_3 = 1 \xrightarrow{C_4} x_4 = 0$$
$$x_4 = 0 \xrightarrow{C_3} x_2 = 0$$

The last implication $x_2 = 0$ conflicts with $x_2 = 1$. As a result, we know that $x_5 = 0$ and $x_1 = 1$ are conflicting assignments for φ. Conflict driven learning generates a clause $(\neg x_1 \vee x_5)$ representing the essential cause of the conflict from the above implications automatically. With this newly added clause (called 'learnt clause'), $x_1 = 1$ directly implies $x_5 = 1$, which avoids the above conflict.

The other is called 'variable state independent decaying sum(VSIDS)', which is a variable selecting heuristic on decision making. The idea behind of this heuristic is that a variable related to many conflicts is relevant to good decision and thus should be selected in the early stage of the decision tree. Each variable is given 'activity', which is initiated to 0 at first. Every time a conflict occurs, activities of variables which is related to the *conflict clauses* are increased by a constant value. A variable which has the highest activity is selected as the next decision variable. After a decision is made, activities of all the variables are decreased by a constant factor. With this heuristic, variables related to recent conflicts have higher activities, and are likely to be chosen.

Both of those heuristics change the SAT solver's internal state. After backtracking from conflict or decision making, the SAT solver accumulates the information of the cause of conflicts. So, the SAT solver performs better in the second execution than in the first run.

3. The Proposed Method

Using modern SAT solver, ATPG problem can be efficiently solved, especially for "hard-to-prove" redundant faults. However, existing methods takes hundreds of seconds for completing ATPG for large ITC99 benchmarks. This section proposes further accelerating technique, which scales well for large designs.

Most of all existing SAT based ATPG algorithms generate one CNF-formula for one fault. In the case of SAT based ATPG, CPU time for generating a CNF formula is comparable to CPU time for solving the SAT problem. For example, the total CPU time for generating CNF formulas for all the faults of C7552 of ISCAS85 benchmark is about 2.40 second on some machine. On the other hand, the total CPU time for solving SAT problems for all the fault of C7552 is 0.98 second on the same machine. That means ATPG is relatively easy problem for SAT solver, however, that also means accelerating techniques for general SAT solvers may not effective. So other approaches are needed for further acceleration.

One relevant feature of ATPG problem is that even for different faults, the good circuit is the same. So, roughly speaking, we are solving similar problems many times. One idea utilizing this feature is to share a CNF-formula for more than one faults. Of course, problems for different faults are also different, so we need some techniques to handle them with one CNF formula. Before treating a general case, consider more easy case. The simplest case is that treats faults of the same location. Let f_0 and f_1 be the stuck-at-0 fault and the stack-at-1 fault of edge (connection) f. Obviously, the conditions that the fault effects propagate to any of the POs are the same. Only the fault activation condition is different ($f = 1$ for f_0, and $f = 0$ for f_1). In this case, we can easily treat those two ATPG problems with one CNF formula. This formula (say φ) is very similar to the CNF formula derived from Fig. 1. In the modified formula φ, however, the fault value is not fixed. Let v_f be the variable in φ representing the value of f in the faulty circuit. Solving the satisfiability of φ with $v_f = 0$ derives the test pattern for f_0, and solving the satisfiability of φ with $v_f = 1$ derives the test pattern for f_1. An important point is that SAT solver does need to change CNF formula with single variable assignments. So we can use the same CNF formula for those two faults.

A slightly complex case is that treats faults in the same FFR. The condition that the fault effect propagates to the root of the FFR is that values of all the side inputs are set to the non-controlling value. And the condition that the fault effect at the root of an FFR propagates to any of the POs is the same for all faults in the same FFR. So, we construct a CNF formula which detects the fault propagation condition from the root of an FFR. The fault propagation condition inside FFR and the fault activation condition is simply represented with single variables assignments. So, we can also use the same CNF formula for faults in the same FFR.

More general case, we cannot treat multiple faults with one

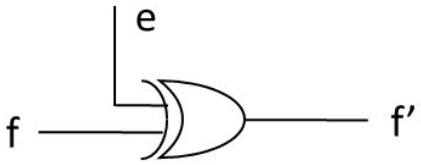

Fig. 2 Fault injection circuit.

simple CNF formula. So, we introduce "fault injection" variables to switch the ATPG problem from one fault to another fault. Let $f_1, f_2, \ldots, f_k$ be the faults to be handled. We introduce variables $e_{f_1}, e_{f_2}, \ldots, e_{f_k}$ to control the fault injection situation. $e_{f_1} = 1$ means the fault f_1 is the subject of detection. For that purpose, a fault injection circuit (**Fig. 2**) is inserted at each fault site. If $e = 0$, the output of the fault injection circuit f' is equal to the input of the circuit f. On the other hand,if $e = 1$, f' is inverted against f. With this modification, we can generate one CNF-formula for any set of faults. Solving SAT problem with $e_{f_1} = 1$ (other fault injection variables are all 0) derives a test pattern for the fault f_1.

This technique is very general and has no restrictions on the fault set. However, increasing the size of a fault set has negative effect for SAT solving, since the number of variables and clauses also increase. So we choose faults in the same MFFC (maximal fanout free cone) as a fault set for CNF generation. From MFFC's definition, it is obvious that the fault effect of any faults in the same MFFC must pass through the root of the MFFC, and the fault propagation conditions after the root of any faults are exactly the same. This leads the CNF sharing efficient. Of course, this is just a heuristic, so there might be another heuristic of grouping fault set for CNF generation.

The CNF sharing technique for faults in the same FFR and the above one is mutually dependent, so those two techniques can be used simultaneously. Rough sketch of the CNF sharing algorithm is as follows:

```
input : a MFFC M

R ← set of FFRs in the MFFC M;
Insert fault injection circuit at each output of r ∈ R;
Generate CNF formula φ for the composed circuit;

for r ∈ R do
    F_r ← set of faults in the FFR r;
    for f ∈ F_v do
        α ← fault activation condition for f;
        β ← fault propagation condition for f;
        Solve φ with α ∧ β ∧ e_v = 1;
    end
end
```

Algorithm 1: CNF sharing algorithm

Consider a fragment of a circuit shown in **Fig. 3**. This circuit itself forms a MFFC, since m is the dominator for all vertices. Subcircuits surrounded by dotted lines are FFRs. For this circuit, fault injection circuits are inserted at g and m, which are both root vertices of FFRs. When deriving a test pattern for stuck-at-0 fault at e, the fault activation condition is $e = 1$, and the fault propagation condition is $f = 0$. Let $e_g(e_m)$ be the fault injection variable for $g(m)$. The assumption (set of assignments) for detecting the stuck-at-0 at e is $e = 1 \wedge f = 0 \wedge e_g = 1 \wedge e_m = 0$.

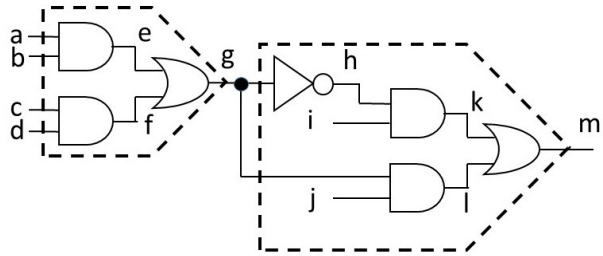

Fig. 3 A circuit example.

Similarly, the assumption for detecting the stuck-at-1 at h is $h = 0 \wedge i = 1 \wedge l = 0 \wedge e_g = 0 \wedge e_m = 1$.

With this algorithm, generating one CNF formula is required for a set of faults in the same MFFC, which obviously reduce the CNF generation time. Furthermore, this algorithm has another side effect that previously generated learnt clauses accelerate the remaining problems. When using MFFC as a criteria of fault grouping, the fault propagation conditions after the root of the MFFC are the same. That means learnt clauses related to the fault propagation conditions for one fault f are also useful for another fault g, since the fault propagation conditions after the root of the MFFC are the same.

Modern SAT solvers utilize VSIDS (Variable State Independent Decaying Sum) heuristics for decision making, which is also effective with CNF sharing because of the similar reason. A good variable for one fault f is likely a good variable for another fault g in the same MFFC.

Without CNF sharing, we have to discard the learnt clauses and have to forget activities of variables, since CNFs for different faults are different even though they are very similar.

4. The Experimental Results

To evaluate the proposed algorithms, experiments using benchmark circuits have been done. The following algorithms are compared:

single: Choose a fault at a time, do ATPG using basic algorithm described in Section 2.

mffc: Choose faults in the same MFFC as one group, do CNF sharing described in Section 3.

For all the experiments, ATPG for all the representative faults were solved. No fault dropping using fault simulation were used.

Benchmark circuits are taken from ISCAS85, ISCAS89, and ITC99. The results of some of the circuits are omitted because the CPU time is too small. CPU is Intel Core i7-2600 (3.40 GHz, 16 GB memory) running FreeBSD 10.3-RELEASE-p3. Compiler is clang-3.4. The program is single thread and does not use multicores. The original hand-made SAT solver is used[*1]. **Table 1** shows the results. The first column ('circuit') denotes the circuit's name. The second and the third columns ('#MFFC' and '#FFR') denote the number of MFFCs and FFRs, respectively. The fourth to the sixth columns ('all', 'det.', and 'red.') denote the number of all faults, detected faults, and redundant faults, respectively. The rest of the columns denote the total CPU time in seconds. The numbers in brace denote the speed-up ratio against **single**.

[*1] The algorithm is very similar to MINISAT-2.2.

Table 1 The comparison among the algorithms.

circuit	#MFFC	#FFR	all	det.	red.	single	mffc	
C432	95	96	524	520	4	0.07	0.03	(2.33)
C499	91	91	758	750	8	0.17	0.05	(3.40)
C880	121	151	942	942	0	0.09	0.04	(2.25)
C1355	91	291	1,574	1,566	8	0.54	0.17	(3.18)
C1908	193	410	1,879	1,870	9	0.71	0.28	(2.54)
C2670	284	594	2,747	2,630	117	0.88	0.38	(2.32)
C3540	398	601	3,428	3,291	137	2.56	1.08	(2.37)
C5315	502	929	5,350	5,291	59	1.89	0.77	(2.45)
C6288	1,488	1,488	7,744	7,710	34	30.85	12.01	(2.57)
C7552	623	1,408	7,550	7,419	131	3.61	1.7	(2.12)
ave. (iscas85)								(2.55)
S1196	180	187	1,242	1,242	0	0.17	0.08	(2.13)
S1238	189	197	1,355	1,286	69	0.20	0.09	(2.22)
S1423	246	259	1,515	1,501	14	0.25	0.11	(2.27)
S1488	101	101	1,486	1,486	0	0.11	0.07	(1.57)
S1494	101	101	1,506	1,494	12	0.11	0.07	(1.56)
S5378	780	1,057	4,551	4,511	40	0.72	0.35	(2.06)
S9234	820	1,263	6,927	6,475	452	2.53	1.00	(2.53)
S13207	1,648	1,676	9,815	9,664	151	2.67	0.89	(3.00)
S15850	1,731	2,202	11,725	11,336	389	5.98	2.08	(2.95)
S35932	4,121	7,343	39,094	35,110	3,984	5.4	1.51	(3.58)
S38417	4,384	6,311	31,180	31,015	165	10.14	3.31	(3.06)
S38584	4,689	5,676	36,303	34,797	1,506	7.04	2.08	(3.38)
ave. (iscas89)								(2.53)
b14	2,129	2,708	22,802	22,646	156	42.03	17.04	(2.47)
b15	2,645	2,872	21,988	21,261	727	69.45	37.70	(1.84)
b17	8,897	9,657	76,625	74,667	1,958	225.49	65.48	(3.45)
b18	31,237	34,387	264,267	262,607	1,660	1,317.90	426.22	(3.09)
b19	63,606	69,723	533,588	529,919	3,669	2,996.89	912.20	(3.29)
b20	4,188	5,157	45,459	45,140	319	137.45	152.58	(2.61)
b21	4,143	5,125	46,154	45,776	378	141.51	51.37	(2.75)
b22	6,347	7,633	67,536	67,192	f 344	202.75	74.40	(2.73)
ave. (itc99)								(2.78)
ave. (total)								(2.60)

Through the experiments, all the ATPG problem was completely solved, i.e. no aborted faults. The number of generated patterns is exactly the same as the number of detected faults, and thus omitted.

CNF sharing algorithm works well for all the circuits. 'mffc' is almost always 2.5 times faster than 'single'. Speed-up ratio is almost the same for all the benchmarks. This is because that the average MFFC size is similar through the circuits (5 – 10 faults per one MFFC).

Table 2 shows the breakdown of CPU time. The second and third columns ('CNF gen.') denote total CPU time for CNF generation in 'single' and 'mffc' mode. The fourth to sixth columns ('SAT') denote total CPU time for SAT solving in 'single' mode and 'mffc' mode [*2]. The program used for the sixth column ('mffc(no-learn)') is slightly modified. In this modified program, all the learnt clauses previously generated by other invocations of SAT solving are discarded before each new invocation of SAT solving. Also, variable activities for VSIDS are cleared. From the results of Table 2, it is obvious that CNF sharing is effective not only for reducing the CNF generation time but also for reducing the SAT solving time significantly. Especially for large circuits, the effect of learnt clauses reusing is obvious. It is worth noting that 'mffc(no-learn)' is almost always larger than 'single', since the number of variables and the number of clauses of the CNF formula of the same fault is always larger than 'single' mode. With learning mechanism, however, the CPU time becomes equal or smaller, since the fault propagation conditions for faults within the same MFFC are almost the same except the conditions inside the MFFC. That results support the MFFC grouping heuristic.

[*2] The rest of the CPU time is mainly spent by generating test vectors from SAT results.

Table 2 The breakdown of CPU time.

	CNF gen.		SAT		
circuit	single	mffc	single	mffc	mffc (no-learn)
C432	0.05	0.01	0.01	0.02	0.03
C499	0.09	0.01	0.08	0.03	0.07
C880	0.06	0.01	0.02	0.02	0.03
C1355	0.38	0.02	0.13	0.12	0.18
C1908	0.49	0.07	0.15	0.18	0.36
C2670	0.58	0.05	0.21	0.26	0.66
C3540	1.70	0.25	0.57	0.67	1.34
C5315	1.20	0.15	0.49	0.49	0.86
C6288	7.12	1.50	22.72	9.98	25.71
C7552	2.40	0.34	0.98	1.11	2.44
S1196	0.12	0.02	0.04	0.03	0.04
S1238	0.13	0.03	0.04	0.03	0.05
S1423	0.19	0.04	0.06	0.05	0.08
S1488	0.07	0.02	0.03	0.03	0.06
S1494	0.07	0.02	0.03	0.03	0.05
S5378	0.50	0.12	0.12	0.16	0.21
S9234	1.68	0.29	0.57	0.50	0.83
S13207	1.84	0.31	0.52	0.39	0.66
S15850	4.20	0.72	1.10	0.95	1.66
S35932	4.38	0.49	0.53	0.72	1.21
S38417	7.69	1.38	1.52	1.37	2.37
S38584	5.54	0.96	0.85	0.69	1.08
b14	24.59	4.40	13.17	10.32	40.65
b15	37.70	5.74	25.57	11.05	24.62
b17	127.14	20.54	77.33	34.58	64.47
b18	619.09	102.11	590.25	274.62	695.87
b19	1,661.70	258.28	1,077.54	538.29	1,214.18
b20	76.87	12.32	46.79	33.48	106.57
b21	81.02	12.49	46.12	31.86	115.76
b22	116.76	18.70	66.10	45.86	118.49

Table 3 Comparison with the existing methods.

circuit	ours	SPIRIT	TIGUAN	PASSAT	TG
C1908	0.01	0.23	0.95	0.64	0.01
C2670	0.04	0.42	2.60	0.91	0.11
C3540	0.06	0.20	5.17	3.83	0.02
C5315	0.03	0.20	3.65	1.44	0.01
C6288	0.06	0.36	7.61	6.57	0.01
C7552	0.12	0.58	5.83	3.31	0.53
s9234	0.21	1.05	6.47	3.53	0.61
s13207	0.25	2.78	6.99	3.64	0.21
s15850	0.34	4.13	12.68	9.18	0.93
s35932	0.25	5.25	17.28	2.96	2.95
s38417	0.63	10.98	23.16	4.21	2.28
s38584	0.46	14.75	22.23	4.84	1.76
b14	3.00	7.34	23.10	19.00	2.68
b15	6.03	52.03	66.82	24.00	27.00
b17	24.10	262.30	252.40	142.00	248.79
b18	110.33	1,555.74	706.92	1,350.00	920.01
b20	6.60	24.52	70.60	56.00	9.93
b21	7.90	39.66	73.46	59.00	8.01
b22	9.32	156.00	90.97	95.00	18.56

The next experiments were done for real setting comparing existing methods. In this experiments, the following flow was used:

(1) Do fault simulation with randomly generated patterns. The simulation loop ends if successive 4 loops (64 patterns per loop, so $64 \times 4 = 256$ patterns) do not find any faults.

(2) Do **mffc** for remaining faults. Every time a new test pattern is found, Fault simulation with the pattern is invoked. Detected faults are excluded from the fault list.

We compare the results with the following existing methods:
SPIRIT [6]: Structural based ATPG
TIGUAN [7]: Thread parallel ATPG
PASSAT [8], [9]: SAT based ATPG
TG [10]: Hybrid (Structural and SAT based) ATPG
The CPU times for the existing methods are taken from [10]. **Table 3** shows the results. All the numbers are CPU times in second.

Since CPU times of existing methods are taken from literature, the background conditions (CPU, OS, compiler, etc.) are not equal. Our results, however, outperform for almost all the circuits [*3]. The difference is significant especially for large circuits (s35932, s38417, s38584, b15, b17, b18, b20, b21, b22). These results show that our method is very effective for large designs, and also show that a pure SAT based ATPG approach (not using structural implication techniques) achieves good performance. SAT based ATPG techniques are easily extendable with adding extra constraints provided they are represented by a CNF formula. So, these results are very encouraging for developing more complicated ATPG algorithms regarding extra constraints.

5. Conclusion

This paper proposes an accelerating technique for SAT based ATPG, which is called 'CNF sharing'. The main idea behind this technique is representing more than one test generation problems as one CNF formula with introducing control variables. CNF sharing reduces not only CNF generation time but also SAT solving time significantly. Experimental results show that the proposed algorithm runs faster than the existing one more than 3 times faster. The proposed algorithm fits to modern efficient SAT algorithms very well, and can be a good base algorithm for more complicated ATPG problem such as a problem for transition fault with signal transition activity constraints.

[*3] For example, our results of larger circuits (b17, b18) are over 10 times faster than others. This gap is far beyond the difference of nowadays machine performance.

References

[1] Larrabee, T.: Test pattern generation using Boolean satisfiability, *IEEE Trans. Computer-Aided Design of Integrated Circuits*, Vol.11, No.1, pp.4–15 (1992).

[2] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proc. 38th Design Automation Conference* (2001).

[3] Zhang, L. and Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, *Proc. IEEE International Conference on Computer Aided Design* (*ICCAD 2001*) (2001).

[4] Marques-Sliva, J. and Sakallah, K.: GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Trans. Computer-Aided Design of Integrated Circuits*, Vol.48, No.5, pp.506–521 (1999).

[5] Eén, N. and Sörensson, N.: An entensible SAT solver, *Proc. International Conference of Theory and Applications of Satisfiability Testing*, pp.502–518 (2004).

[6] Gizdarski, E. and Fujiwara, H.: SPIRIT: A highly robust combinational test generation algorithm, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.21, No.12, pp.1446–1458 (online), DOI: 10.1109/TCAD.2002.804387 (2002).

[7] Czutro, A., Polian, I., Lewis, M., Engelke, P., Reddy, S. and Becker, B.: TIGUAN: Thread-Parallel Integrated Test Pattern Generator Utilizing Satisfiability ANalysis, *VLSI Design, 2009 22nd International Conference on*, pp.227–232 (online), DOI: 10.1109/VLSI.Design.2009.20 (2009).

[8] Shi, J., Fey, G., Drechsler, R., Glowatz, A., Hapke, F. and Schloffel, J.: PASSAT: efficient SAT-based test pattern generation for industrial circuits, *Proc. IEEE Computer Society Annual Symposium on VLSI, 2005*, pp.212–217 (online), DOI: 10.1109/ISVLSI.2005.55 (2005).

[9] Drechsler, R., Eggersgluss, S., Fey, G., Glowatz, A., Hapke, F., Schloeffel, J. and Tille, D.: On Acceleration of SAT-Based ATPG for Industrial Designs, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.27, No.7, pp.1329–1333 (online), DOI: 10.1109/TCAD.2008.923107 (2008).

[10] Chen, H. and Marques-Silva, J.: A Two-Variable Model for SAT-Based ATPG, *IEEE Trans. Computer-Aided Design of Integrated Circuits*, Vol.32, No.12, pp.1943–1956 (online), DOI: 10.1109/TCAD.2013.2275254 (2013).

Yusuke Matsunaga received his B.E., M.E. and Ph.D. degrees in Electronics and Communication Engineering from Waseda University, Tokyo, Japan, in 1985, 1987 and 1997, respectively. He joined Fujitsu Laboratories in Kawasaki, Japan, in 1987 and he has been involved in research and development of the CAD for digital systems. From October 1991 to November 1992, he has been a Visiting Industrial Fellow at the University of California, Berkeley, in the department of Electrical Engineering and Computer Sciences. In 2001, he joined the faculty at Kyushu University. He is currently an associate professor of Department of Advanced Information Technology. His research interest includes logic synthesis, formal verification, high-level synthesis and automatic test pattern generation. He is a member of IEEE, ACM and IPSJ.

(Recommended by Associate Editor: *Yoshinobu Higami*)