Instance Generation for SAT-based ATPG

Daniel Tille Görschwin Fey Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{tille,fey,drechsle}@informatik.uni-bremen.de

***Abstract*— Recently, there is a renewed interest in Automatic Test Pattern Generation (ATPG) based on Boolean Satisfiability (SAT). This results from the availability of very powerful SAT solvers that have been developed in the last few years. Studies have shown that SAT-based ATPG tools can clearly outperform classical approaches for hard-to-test faults. While the ATPG problem has to be solved on a circuit, SAT solvers work on Conjunctive Normal Forms (CNFs).**

In this paper the problem to efficiently generate a SAT instance from a circuit is studied. Experimental results on large industrial circuits show the efficiency of the approach.

I. Introduction

After a circuit has been produced, it is important to test its functional correctness. This is usually done by choosing a fault model and then applying test vectors that show the faults, i.e. the computed output vector of the correct and the faulty circuit differ. While easy-to-detect faults are identified by random pattern simulation, for the harder faults *Automatic Test Pattern Generation* (ATPG) is applied. (For a more detailed overview on testing see e.g. [6].)

The classical ATPG algorithms are based on backtracking, like e.g. FAN [4] or PODEM [5]. While ATPG for combinational circuits was considered as a "solved problem" some years ago, due to the increasing complexity of circuits – according to Moore's law – classical algorithms reach their limits.

As one alternative SAT-based ATPG, that was originally proposed in the early 90s (see [7], [12]), was studied. This revisiting is mainly due to the tremendous improvements in SAT solvers in the last 10 years [8], [9], [2]. Recent work has shown that SAT-based ATPG gives very good results and clearly outperforms classical approaches especially for hard faults. In [11] it has been shown that the SAT-based ATPG tool PASSAT is very robust and is also applicable to large industrial circuits. A detailed study on the integration of a SAT-based ATPG engine into an industrial design can be found in [13].

One major problem in general in SAT-based ATPG is that a circuit has to be transformed into a SAT instance, i.e. a CNF. It has already been observed in [3] that there are many ways to encode the instance and depending on the encoding significant differences in run time and memory consumption result.

In this paper an automatic way of generating SAT instances for multi-input gates, as they frequently occur in industrial circuits, is presented. While other approaches, like proposed in [11], [12], map multi-input gates to sequences of two-input gates (and by this generate many additional auxiliary variables in the SAT instance), in our approach this overhead is removed. This results in more compact instances. Experiments show that not only the memory consumption is reduced, but also the run time of SAT-based ATPG was significantly improved.

II. SAT-based ATPG

In the following SAT-based ATPG as introduced in [7], [12], [11] is briefly reviewed. First, it is shown how an ATPG problem is transformed into a SAT problem and second, the usage of a 4-valued logic to encode unknown signal values and signals at high impedance is shown.

A. SAT Encoding

To find a test pattern for a given circuit C and a fault F two steps have to be performed[1]: First, a circuit is generated that is able to verify that a pattern of input values is a test pattern for F in C. (Such a circuit is called *miter* [1].) And second, this circuit is transformed into a Boolean formula.

In a miter the output behavior of two instances of C are compared, where the faulty behavior is modeled in one of the instances. An assignment of Boolean values to $i_1, \ldots, i_n$ (corresponding to the *Primary Inputs* (PIs)) is a test pattern if and only if at least one output value o_j, $j \in \{1, \ldots, m\}$ differs. At least one output value differs if and only if the output o of the miter circuit becomes 1. In this way the ATPG problem is translated into the following propositional logic problem: "Does there exist an assignment to the Boolean variables $i_1, \ldots, i_n$ so that the Boolean variable o becomes 1"?

The propositional problem can be derived directly by transforming the circuit into a *Conjunctive Normal Form* (CNF). The CNF for a single gate is its characteristic function where each signal is identified with a Boolean variable. The function evaluates to true if and only if the variables' assignments are a legal assignment to the gate. The CNF for a circuit is the conjunction of the CNFs for all gates.

To classify the fault, the SAT instance has to be solved, that represents the miter circuit under an additional condition that forces o to 1. This condition guarantees that at least on pair of outputs assume different values. The solving process is done by an arbitrary SAT solver. If the CNF is satisfiable, the

[1] In practice normally both steps are done at the same time. The separation allows a simple presentation.

TABLE I

ENCODING OF THE 4-VALUED DOMAIN

X	Encode		Interpretation
	c_x	c_x^*	
0	0	0	Signal X is 0
1	1	0	Signal X is 1
U	1	1	Signal X is unknown
Z	0	1	Signal X is at high impedance

TABLE II

DISTRIBUTION OF n-INPUT GATES

Circuit	2	3	4	5	6	7	8
p44k	14461	3257	1702	0	0	0	85
p80k	43487	9916	5167	0	0	0	0
p88k	48594	4162	549	0	0	0	0
p99k	47608	5191	338	2	0	2	15
p177k	84792	6324	1397	0	0	0	0
p462k	203050	14050	2726	461	0	0	0
p565k	376450	18531	1982	0	0	0	0
p1330k	421658	44014	4076	0	0	0	0

satisfying assignment to the variables provides a test pattern. If the solver proves unsatisfiability, the fault is undetectable.

B. Four-valued Logic

As described above, for each signal in the circuit one Boolean variable is needed to encode its Boolean value. However, in industrial circuits it is insufficient to model only Boolean values. In these circuits two additional (non-Boolean) values (U and Z) may occur. Taking these two possible states into account results in a 4-valued logic.

Because SAT is only defined on Boolean formulas, each of the four values has to be encoded by two Boolean variables. There exist 24 possible encodings of this 4-valued logic. A detailed overview is given in [3]. Table I shows the encoding used in the implementation. This encoding works well on circuits with a large portion of Boolean gates. The variable c_x^* indicates whether a signal is Boolean or not.

III. CLAUSE ENCODING

In the last section the direct conversion of a gate into a CNF as used in previous approaches was briefly described. In this section this is considered in detail.

In former approaches only gates with two inputs are considered (cf. [11], [12]). Gates with more than two inputs, in the following called *multi-input gates*, are decomposed.

Table II shows the distribution of multi-input gates in some industrial circuits from NXP Semiconductors. These circuits are discussed in detail in Section V. In each column the accumulated number of AND, NAND, OR, and NOR gates with the respective number of inputs is shown.

In this section different types to model such a multi-input gate are studied. For the sake of convenience, in the following, an n-input gate is called n-*gate*.

A. Two-input vs. Multi-input Gates

In ATPG tools multi-input gates are often broken down to 2-gates which are connected in a cascade (cf. [11]). The formal definition of this construction is given by the following: Let $\odot$ be the gate's function with inputs $i_1, \ldots, i_n$ (where $n > 2$) and output o. Then o is calculated as follows:

$$s_1 := i_1 \tag{1}$$

$$s_j := i_j \odot s_{j-1} \quad \text{for } j = 2, \ldots, n \tag{2}$$

$$o := s_n \tag{3}$$

where $s_2, \ldots, s_{n-1}$ are connections between the 2-gates. This approach is illustrated for a 4-AND gate in Figure 1.

Because of the auxiliary signals (in Figure 1 denoted by t_1 and t_2) there is an overhead of $n-1$ variables in 2-valued and $2 \cdot (n-1)$ variables in the 4-valued logic.

All auxiliary variables can be dismissed, if an n-gate is modeled as one single gate. However, the number of clauses needed to model an n-gate in 4-valued logic grows exponentially. Table III shows the CNF sizes for n-input AND and OR gates. The columns '2-input', 'Multi-input', and 'Bounded' present the numbers for the approach "divide the n-gate into $(n-1)$ 2-gates", "use normal n-gate", and "use bounded multi-input gates", respectively. The bounded multi-input approach is explained in detail in the next section. 'Vars' and 'Cls' give the number of variables and clauses, respectively. Note that the sizes for AND and OR gates are equal to the sizes for NOR and NAND gates, respectively.

TABLE III

CNF SIZES FOR SOME n-GATES

Gate	2-input		Multi-input		Bounded	
	Vars	Cls	Vars	Cls	Vars	Cls
2-AND	6	8	6	8	6	8
3-AND	10	16	8	13	8	13
4-AND	14	24	10	22	10	22
5-AND	18	32	12	39	12	39
6-AND	22	40	14	72	16	47
7-AND	26	48	16	137	18	52
8-AND	30	56	18	266	20	61
2-OR	6	9	6	9	6	9
3-OR	10	18	8	15	8	15
4-OR	14	27	10	25	10	25
5-OR	18	36	12	43	12	43
6-OR	22	45	14	77	16	52
7-OR	26	54	16	143	18	58
8-OR	30	63	18	273	20	68

As basic gates, the CNF sizes of the 2-AND and the 2-OR gates are equal in all three cases. At each input level the number of variables in the 2-input approach grows by four, whereas, in the multi-input approach only two additional variables in each level are needed. So the difference of the two approaches is 12 variables for an 8-gate. However, the number of clauses needed to model an AND gate or an OR gate with more than five inputs exceeds the number needed in the 2-input approach. More than four times the clauses for an 8-OR gate or an 8-AND gate are required. The reason is the exponential growth of needed clauses. To model an n-AND $2^n + n + 2$ clauses are needed, while an n-OR requires $2^n + 2 \cdot n + 1$ clauses.

An example of these two approaches is shown in Figure 2. The 4-AND gate from Figure 1 is converted into a CNF, where

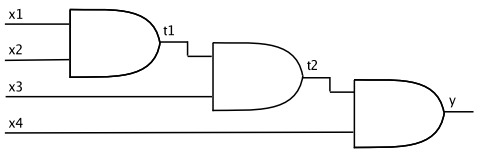

Fig. 1. 4-AND gate modeled by a sequence of three 2-AND gates

$$
\begin{array}{l}
(\overline{x}_1 + \overline{x}_2 + t_1) \cdot (x_1^* + x_2^* + \overline{t}_1^*) \cdot (t_1 + \overline{t}_1^*) \cdot \\
(x_1 + x_1^* + \overline{t}_1) \cdot (x_2 + x_2^* + \overline{t}_1) \cdot (\overline{x}_1^* + \overline{x}_2 + t_1^*) \cdot \\
(\overline{x}_1 + \overline{x}_2^* + t_1^*) \cdot (\overline{x}_1^* + \overline{x}_2^* + t_1^*) \cdot \\
(\overline{t}_1 + \overline{x}_3 + t_2) \cdot (t_1^* + x_3^* + \overline{t}_2^*) \cdot (t_2 + \overline{t}_2^*) \cdot \\
(t_1 + t_1^* + \overline{t}_2) \cdot (x_3 + x_3^* + \overline{t}_2) \cdot (\overline{t}_1^* + \overline{x}_3 + t_2^*) \cdot \\
(\overline{t}_1 + \overline{x}_3^* + t_2^*) \cdot (\overline{t}_1^* + \overline{x}_3^* + t_2^*) \cdot \\
(\overline{t}_2 + \overline{x}_4 + y) \cdot (t_2^* + x_4^* + \overline{y}^*) \cdot (y + \overline{y}^*) \cdot \\
(t_2 + t_2^* + \overline{y}) \cdot (x_4 + x_4^* + \overline{y}) \cdot (\overline{t}_2^* + \overline{x}_4 + y^*) \cdot \\
(\overline{t}_2 + \overline{x}_4^* + y^*) \cdot (\overline{t}_2^* + \overline{x}_4^* + y^*)
\end{array}
$$

(a) 4-AND consisting of 2-ANDs

$$
\begin{array}{l}
(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4^* + y^*) \cdot (\overline{x}_1 + \overline{x}_2 + \overline{x}_3^* + \overline{x}_4 + y^*) \cdot \\
(\overline{x}_1 + \overline{x}_2^* + \overline{x}_3 + \overline{x}_4 + y^*) \cdot (\overline{x}_1^* + \overline{x}_2 + \overline{x}_3 + \overline{x}_4 + y^*) \cdot \\
(\overline{x}_1 + \overline{x}_2 + \overline{x}_3^* + \overline{x}_4^* + y^*) \cdot (\overline{x}_1 + \overline{x}_2^* + \overline{x}_3 + \overline{x}_4^* + y^*) \cdot \\
(\overline{x}_1^* + \overline{x}_2 + \overline{x}_3 + \overline{x}_4^* + y^*) \cdot (\overline{x}_1 + \overline{x}_2^* + \overline{x}_3^* + \overline{x}_4 + y^*) \cdot \\
(\overline{x}_1^* + \overline{x}_2 + \overline{x}_3^* + \overline{x}_4 + y^*) \cdot (\overline{x}_1^* + \overline{x}_2^* + \overline{x}_3 + \overline{x}_4 + y^*) \cdot \\
(\overline{x}_1 + \overline{x}_2^* + \overline{x}_3^* + \overline{x}_4^* + y^*) \cdot (\overline{x}_1^* + \overline{x}_2 + \overline{x}_3^* + \overline{x}_4^* + y^*) \cdot \\
(\overline{x}_1^* + \overline{x}_2^* + \overline{x}_3 + \overline{x}_4^* + y^*) \cdot (\overline{x}_1^* + \overline{x}_2^* + \overline{x}_3^* + \overline{x}_4 + y^*) \cdot \\
(\overline{x}_1^* + \overline{x}_2^* + \overline{x}_3^* + \overline{x}_4^* + y^*) \cdot (x_4 + x_4^* + \overline{y}) \cdot \\
(x_3 + x_3^* + \overline{y}) \cdot (x_2 + x_2^* + \overline{y}) \cdot \\
(x_1 + x_1^* + \overline{y}) \cdot (y + \overline{y}^*) \cdot \\
(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4 + y) \cdot (x_1^* + x_2^* + x_3^* + x_4^* + \overline{y}^*)
\end{array}
$$

(b) 4-AND as one single gate

Fig. 2. Clauses for the 4-valued 4-AND gate

in Figure 2(a) the cascaded 2-AND approach is used and in Figure 2(b) the 4-AND gate was modeled as a single gate. In the multi-input approach the CNF is smaller: Two clauses and four variables less than in the 2-input approach are needed. This corresponds to line three in Table III.

B. Bounded Multi-input Gates

Above, the advantages and drawbacks of modeling multi-input gates for many inputs were described: In the 2-input approach the number of variables grows significantly. In the multi-input approach the number of variables grows slightly, but there is an exponential growth of clauses, which is smaller in the 2-input approach. Up to five inputs the number of clauses is acceptable. Therefore the *bounded multi-input* approach is proposed. This is a combination where the multi-input approach is used, but the number of inputs per gate is limited.

According to Table III the input number is set to five. To model a gate with more than five inputs, too many clauses are required while, on the other hand, the variable saving in gates with less than five inputs are low.

In the bounded multi-input approach, gates with more than five inputs are divided into sequences of 5-gates.

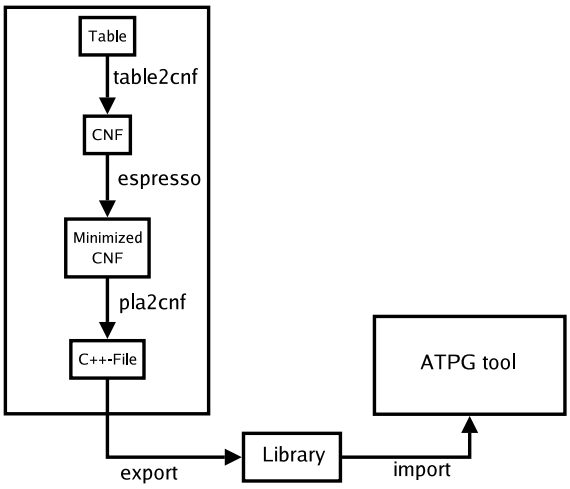

Fig. 3. Clause generation working flow

TABLE IV
EXPERIMENTAL RESULTS

Circuit	2-input		Bounded multi-input	
	Aborted	Run time	Aborted	Run time
p44k	2583	25:11h	0	2:18h
p80k	1	52:24m	1	42:58m
p88k	0	12:13m	0	11:41m
p99k	0	9:07m	0	8:41m
p177k	1337	13:26h	941	10:28h
p462k	155	3:53h	129	3:31h
p565k	0	2:42h	0	2:23h
p1330k	1	5:28h	1	4:58h

IV. CLAUSE GENERATION

Figure 3 shows the general flow of the clause generation. Consider the box on the left side: First, the truth table of a gate's function is created. This table is translated into a CNF using the script *table2cnf*. This CNF is minimized by espresso [10]. Since the Boolean function is small enough, an optimal minimization algorithm can be applied. The script *pla2cnf* converts the minimized CNF into C++-code that adds clauses to the SAT solver. This function is independent of the used SAT solver, since an abstract interface is used.

This work flow is applied once for each gate type and for each number of gate inputs. Each pass creates a function in C++-code which is exported in a library. This library is included in the ATPG tool. For each gate occurring in the circuit that has to be added to the SAT solver, the respective function in the library is called.

V. EXPERIMENTAL RESULTS

The experiments were carried out with an improved version of PASSAT [11]. As SAT solver MiniSat [2] was applied. For each fault, first, the solver was started with a timeout of 5 seconds where the solver is allowed to branch on all variables and, second, if the search was aborted with a timeout, the solver was started with a timeout of 15 seconds where the solver can branch on PIs only (cf. [11]).

TABLE V
INSTANCE SIZES

	2-input				Bounded multi-input			
	Max		Mean		Max		Mean	
Circuit	Vars	Cls	Vars	Cls	Vars	Cls	Vars	Cls
P44k	103,154	339,844	71,933	221,436	101,491	328,732	60,446	209,001
P80k	396,956	1,328,604	8,308	23,328	356,452	1,293,556	7,483	22,697
P88k	96,071	302,362	5,276	15,951	93,133	298,652	5,047	15,676
P99k	35,892	131,511	5,689	16,687	34,640	129,746	5,301	16,139
P177k	730,397	2,492,841	73,506	227,871	702,975	2,460,985	69,908	222,516
P462k	406,847	1,375,884	7,473	22,346	391,113	1,361,732	7,336	22,399
P565k	1,735,442	5,575,293	4,829	15,827	1,705,996	5,536,393	4,663	15,638
P1330k	567,331	1,943,536	21,459	64,170	552,049	1,925,728	20,580	62,929

Table IV shows the results for some industrial benchmarks from NXP Semiconductors Germany GmbH, Hamburg, Germany. All experiments were carried out on a Dual DualCore 64-Bit Xeon system (3.0 GHz, 32 GByte, GNU/Linux). Column 'Circuit' shows the circuit's name. The name provides information on the circuit size, e.g., p565k means the circuit contains about 565,000 gates.

The columns '2-input' and 'Bounded multi-input' show the results of the two approaches for clause encoding, where the columns 'Aborted' and 'Run time' give the number of timeouts during the search[2] and the total run time of the entire ATPG search, respectively.

As can be seen, on all circuits, the results of the bounded multi-input approach are better than the results of the 2-input approach. On some circuits the new approach even yields substantially better results (e.g. p44k) and on some circuits the gain is small (e.g. p88k). This can be explained by considering Table II. Consider the relative number of multi-input gates (with respect to the number of all gates). In circuits with much gain, this number is high. In these cases less variables are needed and therefore the SAT instance can be solved more easily. Analogically, circuits with small gain have an low number of multi-input gates.

Table V provides further insight; an overview on the size of the SAT instances is given. The columns 'Max' and 'Mean' give the largest and the average size, respectively, where the columns 'Vars' and 'Cls' give the number of variables and clauses, respectively.

It can be seen that the bounded multi-input approach generates SAT instances with less variables and (except for p462k) with less clauses than the 2-input approach. Besides reducing the run time this also implies savings in memory requirements.

VI. CONCLUSIONS

In this paper the efficient generation of instances for SAT-based ATPG was studied. With the proposed approach, multi-input gates can be translated very compactly. Experimental results demonstrated the efficiency of the approach. Beside the memory reduction, also run time could be saved.

[2]An abort occurs when the CNF for a fault is not solvable within 20 seconds.

ACKNOWLEDGEMENTS

This work was funded in part by NXP Semiconductors Germany GmbH, Hamburg, Germany, in context of the BMBF project MAYA (01M3172B) and in part by DFG grant DR 287/15-1.

Furthermore, the authors would like to thank Stephan Eggersglüß for helpful discussions and Andreas Glowatz for his support.

REFERENCES

[1] D. Brand. Verification of large synthesized designs. In *Int'l Conf. on CAD*, pages 534–537, 1993.

[2] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[3] G. Fey, J. Shi, and R. Drechsler. Efficiency of multiple-valued encoding in SAT-based ATPG. In *Int'l Symp. on Multi-Valued Logic*, page 25, 2006.

[4] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. on Comp.*, 32:1137–1144, 1983.

[5] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic. *IEEE Trans. on Comp.*, 30:215–222, 1981.

[6] N. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2003.

[7] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

[8] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[9] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[10] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[11] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel. PASSAT: Efficient SAT-based test pattern generation for industrial circuits. In *IEEE Annual Symposium on VLSI*, pages 212–217, 2005.

[12] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD*, 15:1167–1176, 1996.

[13] D. Tille, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel. Studies on integrating SAT-based ATPG in an industrial environment. In *GI/ITG/GMM-Workshop "Testmethods and Reliability of Circuits and Systems"*, 2007.