INVITED PAPER

Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design

In this paper, the author presents a method by which logic synthesis and formal verification can be achieved with a small number of input patterns, if all possible circuit transformations are predetermined.

By Masahiro Fujita

ABSTRACT | In logic synthesis, the search space is infinite in the sense that any number of gates can be connected using any topology to come up with the best circuit under performance and other constraints. In formal verification, to achieve full coverage, an n-input circuit must be checked either explicitly or implicitly using the complete set of 2^n input values. There are, however, situations when the possible circuit topologies are limited. Logic optimization methods, in general, do not change circuit topologies dramatically. Most of them are based on series of local circuit transformations. In this paper, we discuss formal verification of circuits produced by logic synthesis where the search space is limited, and only gates or subcircuits are transformed whereas their interconnections never change. If there are p possible transformations for each gate or subcircuit, and there are m gates or subcircuits in the entire circuit, the number of all possible transformations for the entire circuit is p^m. Logic synthesis with this restriction attempts to find the best circuit among the p^m alternatives. The logic synthesis problem can be formulated as a sequence of incremental SAT problems and the complete set of test patterns can be computed, which detects all possible errors in the synthesized circuit. As long as the search space is limited to the p^m alternatives, such complete set of test patterns can be used for formal verification. The number of test patterns needed in this case is experimentally shown to be very small, e.g., a few hundred, even for circuits having several hundred inputs and several thousand gates. With the complete set of test patterns generated for the circuit transformations, logic synthesis and formal verification are unified in the sense that the small number of test patterns allows for logic synthesis with 100% correctness.

KEYWORDS | Automatic test pattern generation; circuit synthesis; formal verification; logic circuits

I. INTRODUCTION

Logic synthesis tries to generate fully optimized circuits from the given specification, such as logic formulas in the case of combinational circuits, under performance constraints, such as area, delay, power, and so on. The ways to implement the given logic formulas can be infinitely many, as there can be any number of gates interconnected using any topology. Therefore, the search space is infinite in general, and the implementation requires 2^n input patterns for an n-input combinational circuit to ensure its 100% correctness after synthesis. In this paper, we discuss possible unification of logic synthesis and the verification of its results targeting combinational circuits or time frame expanded sequential circuits.

There have been a number of research efforts to unify the verification and synthesis of logic circuits. These exist in various contexts. For example, Mishchenko and Brayton [1] utilize complete flexibility, Jiang *et al.* [2] synthesize from relations rather than functions, Lai *et al.* [3] utilize Boolean matching-based methods, Lin *et al.* [4] apply function decomposition, Lin *et al.* [5] implement the engineering change order (ECO), and Seshia [6] shows a combined approach for both logic verification and logic synthesis via inductive learning. These can be used in logic synthesis and verification for topologically constrained logic circuits and many other logic synthesis techniques using binary decision diagram (BDD) and satisfiability

Manuscript received November 10, 2014; revised April 6, 2015; accepted June 25, 2015. Date of publication October 2, 2015; date of current version October 26, 2015.
The author is with University of Tokyo, Tokyo 113-0032, Japan (e-mail: fujita@ee.t.u-tokyo.ac.jp).

Digital Object Identifier: 10.1109/JPROC.2015.2476472

checking (SAT) for solution derivation. There is an effort to establish a framework for logic synthesis and verification, such as verification interacting with synthesis (VIS) [7] and a system for sequential synthesis and verification (ABC) [8]. Also, there are efforts on topologically constrained logic synthesis, such as [9], which has a similar objective to the work shown below.

There are, however, more restricted and practical situations when the search space for logic synthesis can be considered finite. Most of practically used logic synthesis techniques are based on iterative refinements or optimizations of the circuits. They apply series of local circuit transformations to the current circuits in such a way that transformed circuits become better in terms of the given criteria, such as an area, the delay, power consumptions, and others. In this paper, we deal with such restricted logic synthesis where the synthesized circuits are the result of multiple transformations of each gate or subcircuit. Each gate or subcircuit in the given circuit has a set of possible transformations, and logic synthesis here is to apply such transformations to multiple gates or subcircuits. Each gate or subcircuit may be transformed or not transformed, which is the decision made by logic synthesis.

If there are p transformations defined for each gate or subcircuit, and there are the total of m gates or subcircuits in the given circuit, the total number of possible multiple transformations allowed in the circuit is p^m. In this case, during logic synthesis, only gates or subcircuits are transformed, and their interconnections remain the same. That is, circuit topologies remain the same. Here, logic synthesis just selects the correct and best set of transformations among the p_m choices.

Since the interconnections among gates or subcircuits do not change, the timing characteristics may not change much. If the original circuit satisfies the given performance constraints, the circuits after transformations may also satisfy them. Or the selection of circuit transformations can be based on the performance criteria, so that the resulting circuit is always better. So the main issue is the functional correctness of the synthesized circuits. The logic synthesis problem in the above context can be formulated as a series of SAT problems and efficiently solved by incremental SAT solvers.

First, the set of all possible transformations is implicitly defined as logic formulas with "parameter variables." Depending on the values of parameter variables, different transformations are applied to the circuit. So the goal is to find the values of parameter variables which make the circuit logically equivalent to the specification, which is given separately. In order to solve this problem, counterexamples, which are the input patterns detecting incorrectness of some of the transformations, are repeatedly generated until no more counterexamples exist.

This series of SAT problems is essentially an incremental SAT problem where later SAT problems are always supersets of conjunctive normal form (CNF) formulas, compared to the previous ones. When the problem becomes unsatisfiable (UNSAT), which means there exist no more counterexamples, the set of input pattens, which corresponds to the generated counterexamples, is a complete set of test patterns. If the synthesized circuit is correct under these input patterns, it is guaranteed to be correct for all of 2^n input patterns where n is the number of primary inputs. That is, any set of transformations which are correct under this set of input patterns corresponds to a correctly synthesized circuit. Depending on criteria related to performance and other metrics, designers can choose one among all of these solutions. On the other hand, if there does not exist a set of transformations, which are correct for all of the input patterns, this is a mathematical proof that the function cannot be implemented using this topology and the given set of transformations.

The above idea was originally introduced as counterexample-guided inductive synthesis (CEGIS) in [13]. The problem can be naturally formulated as a quantified boolean formula (QBF), but it can be solved through repeated application of SAT solvers, instead of QBF solvers, which was first discussed under field-programmable gate array (FPGA) synthesis in [14] and in program synthesis in [13]. Janota *et al.* [15] discuss the general framework on how to deal with QBF only with SAT solvers. In repeated application of SAT solvers, constraints are incrementally or monotonically increasing, and so incremental SAT solvers should work well, which is also discussed in [13].

The number of input patterns, which can completely check the correctness of the synthesized circuit, is experimentally shown to be very small, compared to 2^n patterns required in general logic synthesis for a circuit with n primary inputs. For example, as shown later, in s13207 in Table 1, for a combinational circuit with 700 inputs, 4600 gates, and two transformations defined for each gate, the number of input patterns required to guarantee the 100% correctness of the circuit (that is, to perform formal verification) is only 414, which is surprisingly small, compared to the complete set of 2^{700} input combinations.

The observation that the number of test patterns and iterations for synthesis is small has been made and analyzed in the synthesis of bit-vector programs [10]. Here it is also the case that a small number of test patterns finds all wrong transformations and can identify the correct ones.

With such small sets of input patterns needed to check the 100% correctness, logic synthesis and formal verification of combinational circuits can be discussed in a uniform and simple way, as we need to check only these input values for any set of transformations to ensure the complete correctness.

The paper is organized as follows. Section II shows the proposed method by which all sets of transformations can be implicitly represented. There are p^m such sets, where p

is the number of possible transformations for each gate or subcircuit and m is the number of gates or subcircuits. Section III describes the proposed logic synthesis and verification method. Experimental results follow in Section IV, showing the effectiveness of the proposed approach. Section V contains concluding remarks.

II. REPRESENTATION METHOD FOR TRANSFORMING LOGIC FORMULAS

In our formulation, transformations of gates or subcircuits are defined in terms of the possible resulting logic functions after the transformations. In our notation, x variables are called parameter variables and, depending on their values, transformations are defined as the resulting logic functions.

Fig. 1 shows two example transformations. Fig. 1(a) is a set of transformations where inputs and outputs of gates may change their polarities. If the original gate in the circuit is a two-input AND gate, there are eight possible transformation results, which are determined by the combination of values of x_1, x_2, and x_3. Each value combination of x_1, x_2, and x_3 determines one particular transformation, and, by explicitly or implicitly enumerating all value combinations, all possible transformations are explored.

The resulting logic functions realized at the output c for the transformations in Fig. 1(a) can be defined as a function of a, b and x_1, x_2, x_3, where a, b are original inputs of the AND gate to be transformed and x_1, x_2, x_3 are the parameter variables that define particular transformations

$$((a \oplus x_1) \wedge (b \oplus x_1)) \oplus x_3$$

where $\oplus$ means exclusive-OR operation.

Fig. 1(b) defines most general transformations where all possible logic two-input functions are defined using four parameter variables x_1, x_2, x_3, and x_4. Basically, x_1, x_2, x_3, and x_4 are the values of the truth tables of the resulting logic functions after transformations. By explicitly or implicitly enumerating all value combinations of x_1, x_2, x_3, and x_4, all possible truth tables can be generated, which means all logic functions with two inputs can be explored.

The resulting logic functions realized at the output c for the transformations in Fig. 1(b) can be defined as a function of a, b and x_1, x_2, x_3, x_4, where a, b are original inputs of the AND gate to be transformed and x_1, x_2, x_3, x_4 are the parameter variables that define particular transformations

$$((\neg a \wedge \neg b) \rightarrow c = x_1) \wedge ((a \wedge \neg b) \rightarrow c = x_2)$$
$$\wedge ((\neg a \wedge b) \rightarrow c = x_3) \wedge ((a \wedge b) \rightarrow c = x_4)$$

where $\rightarrow$ means an implication operation.

Depending on x_1, x_2, x_3, x_4 values, the function selects an appropriate value from the truth table. Therefore, with the logic functions and appropriate values for x_1, x_2, x_3, x_4, any logic function with two inputs can be represented.

In our proposed method, as can be seen from the above examples, the logic functions representing all resulting logic functions by the given transformations, are defined first. Once they are defined, each gate or subcircuit is replaced with those logic functions. The resulting circuit can represent all possible combinations of transformations of gates or subcircuits by explicitly or implicitly enumerating all possible values of the parameter variables.

Note that, if there are m gates or subcircuits, there are m of $x_1, x_2, \ldots$ in the entire circuit with transformations. All of them are enumerated in order to deal with all possible "multiple" transformations.

Also note that so far only the transformations of gates have been shown. But clearly, it is straightforward to define the logic functions of transformations for gates or subcircuits consisting of multiple interconnected gates rather than single gates. Moreover, a subcircuit can have disjoint sets of gates when there is no connection between the disjoint sets, if that is preferred.

Now, when the given circuit is expanded with the logic functions of transformations, the SAT-based analysis can be applied by translating the circuit into CNF. This process is the same as the one used for translating normal circuits. In Section III, the proposed logic synthesis and verification method with small numbers of input patterns is demonstrated by applying the SAT-based analysis.

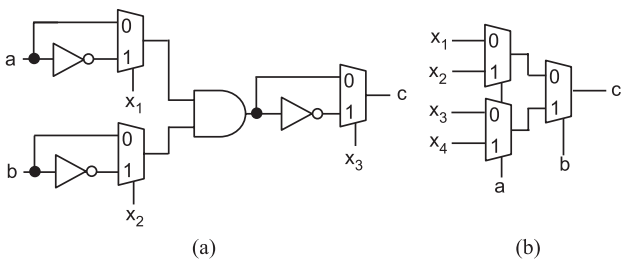

Fig. 1. *Examples of transformations of gates. (a) Transformations of input/output polarities. (b) Transformations for all possible logic functions.*

III. THE PROPOSED LOGIC SYNTHESIS METHOD

The proposed logic synthesis is to find appropriate values for parameter variables which define gate or subcircuit transformations in such a way that the resulting entire circuits become logically equivalent to their specifications. That is, with the appropriate values of parameter variables, i.e., there exist such values, for all primary input values,

the outputs of the entire circuits are equal to the outputs of the specifications. This can be formulated naturally as a two-level QBF where the existential quantifiers are followed by the universal quantifiers. This formulation is explained in detail in the following. The two-level QBF is solved by repeatedly applying SAT solvers rather than QBF solvers in our proposed method.

The proposed logic synthesis and verification flow is shown in Fig. 2. First, from the given circuit and its associated transformations of gates or subcircuits, each gate or subcircuit is replaced with the logic functions defined from the given transformations, as discussed in Section II.

Then, the complete set of test patterns is generated using the method to compute the test patterns detecting all functionally wrong transformations, which is based on the incremental SAT formulation shown next.

Here "complete" means that if a transformed circuit is correct under all of these test patterns, it is guaranteed to be correct for all possible input patterns, which is 2^n where n is the number of primary inputs of the given circuit.

In the third step, transformations which are correct under these complete sets of test patterns are searched. This is a normal SAT problem, and depending on cost functions in terms of performance and other criteria, appropriate transformations can be picked up from the solutions. For example, a solution which has the minimum number of transformed gates or subcircuits may be taken.

So the central part of the proposed method is to generate the complete set of test patterns, which will be discussed below. An intuitive explanation is given, followed by more mathematical discussions.

The basic idea is to look for a counterexample for some transformations. That is, the identified counterexamples generate wrong primary output values if those transformations are applied. This is a simple SAT problem specifying whether this is an input pattern under which the primary output values with some transformations are wrong. The input pattern will be the first pattern in the complete set of test patterns. Once a counterexample is generated, additional constraint specifying that the circuit must output correct values with that counterexample is added to the original SAT formula, and the SAT solver is applied to the resulting formula in the next iteration.

This process is repeated until there are no more counterexamples generated, which means that there is no input pattern which can generate wrong values at the output under any transformation. That is, the set of collected counterexamples is a complete set of test patterns for the entire circuit with the given transformations. Therefore, any transformation which can generate correct primary output values for all of these test patterns is functionally 100% correct. This is again a SAT problem, and if there is no such transformation, that is, the SAT solver returns UNSAT, it is a mathematical proof that there is no way to synthesize circuits with the given sets of transformations.

Now we show the proposed method in a more mathematical and procedural form. This is essentially equivalent to the CEGIS algorithm discussed in [13], but it is targeted to logic circuits. For ease of explanation, we assume the number of primary outputs of the given circuit to be one. It is straightforward to extend the method below for the cases of multiple primary outputs.

1) Let $\mathrm{circuit}(X, \mathrm{In})$ be the formula corresponding to the given circuits with the given transformations encoded, as discussed in Section II. Here X is the set of all parameter variables for the transformations, and In is the set of primary inputs of the circuit.
2) Let $\mathrm{spec}(\mathrm{In})$ be the formula corresponding to the specification.
3) Let k be the number of test vectors in the generated set of test patterns, initialized to zero, and let TestSet be empty initially.
4) Let Target be $\mathrm{circuit}(X, \mathrm{In}) \neq \mathrm{spec}(\mathrm{In})$ where X and In are variables whose values are to be identified by a SAT solver.
5) Check if Target is satisfiable. This is a normal SAT problem.
6) If it is SAT, let $k = k + 1$ and denote a solution as (x_k, in_k). Let $\mathrm{TestSet} = \mathrm{TestSet} \cup \mathrm{in}_k$ and $\mathrm{Target} = \mathrm{Target} \wedge (\mathrm{circuit}(X, \mathrm{in}_k) = \mathrm{spec}(\mathrm{in}_k))$, and go back to 5).
7) If UNSAT, check if the following formula is satisfiable: $(\mathrm{circuit}(X, \mathrm{in}_1) = \mathrm{spec}(\mathrm{in}_1)) \wedge (\mathrm{circuit}(X, \mathrm{in}_2) = \mathrm{spec}(\mathrm{in}_2)) \wedge \cdots \wedge (\mathrm{circuit}(X, \mathrm{in}_k) = \mathrm{spec}(\mathrm{in}_k))$. If SAT, then for any solution (x, in), x is a correct set of transformations leading to an equivalent circuit. Thus, this formula implicitly represents the

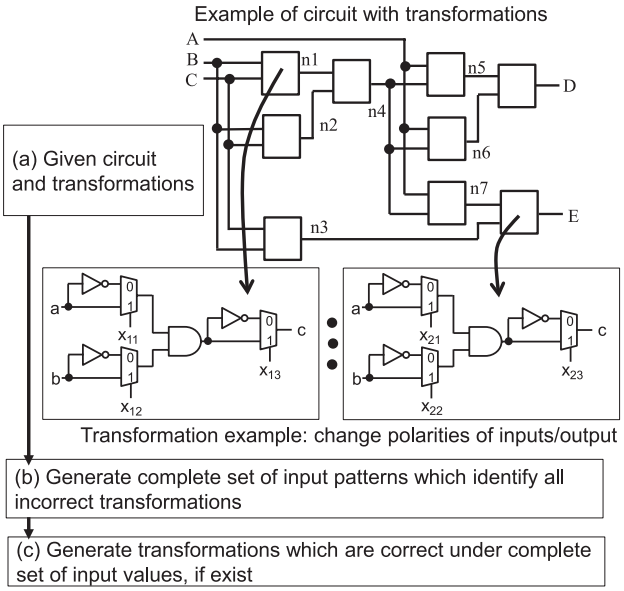

Fig. 2. *Flow of the proposed logic synthesis.*

set of all correct circuits. Otherwise, there does not exist a correct set of transformations.

In the above, $\exists X.\forall \mathrm{In}.\mathrm{circuit}(X, In) = \mathrm{spec}(\mathrm{In})$ is the target QBF to be solved. Instead of applying QBF solvers directly to the problem, SAT solvers are applied repeatedly until no more counterexamples can be generated. TestSet keeps the set of counterexamples.

Theorem: TestSet is complete.

Proof: Suppose it is not complete. Then, there exists a circuit $C(x*, \mathrm{In})$ such that $C(x*, \mathrm{in}) = \mathrm{spec}(\mathrm{in}), \forall \mathrm{in} \in \mathrm{TestSet}$, and an input in$*$ such that $C(x*, \mathrm{in}*) \neq \mathrm{spec}(\mathrm{in}*)$. This implies that $(x*, \mathrm{in}*)$ is a satisfying assignment to $\mathrm{Target}_k(X, \mathrm{IN})$. This contradicts that Target_k is UNSAT. ■

The above procedure is illustrated as a flowchart in Fig. 3. The efficiency of the proposed method fully depends on how large k is. If k is very large, it is practically an unsolvable problem. But if k is small, very efficient synthesis and verification can be expected. As shown in Section IV, k is normally very small, compared to the numbers of all possible input patterns. For example, a circuit having 700 primary inputs and 4633 gates (combinational part of s13207 of ISCAS89 benchmark circuits) needs only 414 test patterns when transformation is to change output polarities of gates. This is significantly smaller than the number of all possible input patterns, which is 2^{700}.

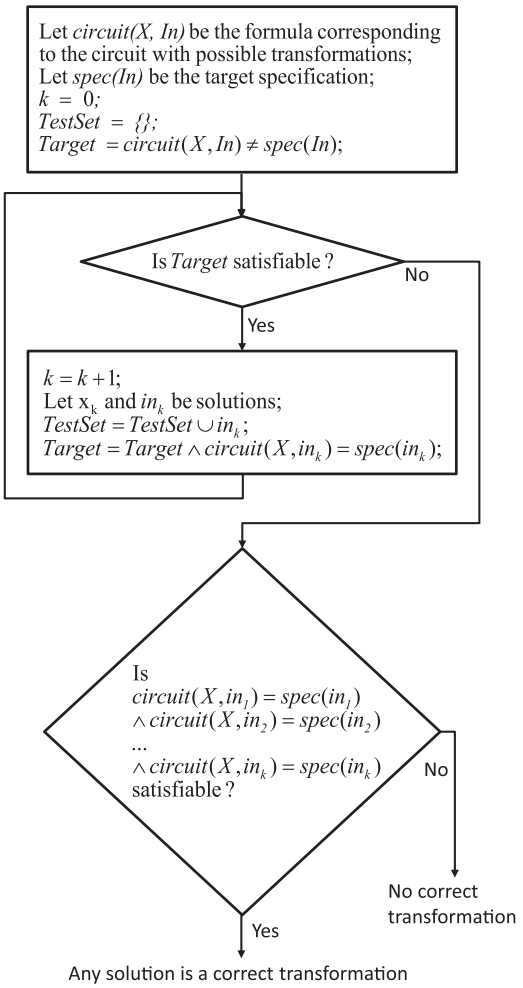

Fig. 3. ***Flowchart of the proposed method.***

The reason why such a small number of test patterns can be a complete set is because of the fact that the numbers of possible multiple transformations are p^m where p is the number of transformations for each gate or subcircuit and m is the number of gates or subcircuits. Although this is exponentially large, it is also finite. If there is no information on the internal structure of the circuit, we do need to check all of 2^n where n is the number of primary inputs, as there are infinitely many ways to implement the circuit. The logic synthesis and verification problem in this paper has only finitely many ways of implementation, which are defined as transformations. Although theoretically this does not indicate that a small number of test patterns is sufficient, practically there is a good chance that a small number of test patterns can identify all wrong transformations. In any case, we can discuss this using the experimental results, which are discussed next.

As can be easily seen from step 6) of the above procedure, the SAT problems to be solved are incremental; each one has more constraints compared to the previous ones. This means that any learned clauses obtained from the previous SAT problems are valid in the following SAT problems. That is, a SAT solver can start searching from the point where the previous SAT problem causes conflict or backtrack. Therefore, the sequence of SAT problems can be considered as a single incremental SAT problem. In this case, the problem starts with initial constraints and adds more constraints based on the counterexamples, until the SAT problem finally becomes UNSAT. In the experiments below, we utilize this property in such a way that all learned clauses obtained in the previous SAT calls are saved to skip previous conflicts.

There can be multiple or typically many solutions which satisfy the final constraints above. From the viewpoint of logic synthesis and optimization, the performance of the entire circuit may change up to some amount depending on which solutions are used for replacements. This is particularly true when we simultaneously replace a number of gates or subcircuits. Although we have not yet explored this much, here are some basic discussions. For logic optimization targeting delays, the proposed method can be used to identify alternative logic or paths for the current timing critical paths. When a critical path is given, that path is cut at the output of some gate and the alternative logic gates are inserted with a lookup table (LUT). Then, the problem is to find appropriate logic

contents for the LUTs which make an entire circuit logically correct even if the critical path is cut. This is a typical procedure for timing optimization, but in our formulation, we can insert multiple LUTs that are simultaneously programmed, which may make the optimization more successful. A similar approach can be applied to the area optimization as well. In such cases, alternative subcircuits are explored for area reductions. Also, polarities of internal gates sometimes may have to be adjusted for efficient implementations, and transformations with polarities shown above may be useful. For ECO or logic debugging, the proposed method can be directly applied with area and timing constraints in mind. Further research can be done in these directions.

IV. EXPERIMENTAL RESULTS

The proposed method has been implemented in two ways. The first implementation is on top of the logic synthesis and verification tool ABC [8], and the second is an independent software using AIGER tools [11] and the PicoSAT solver [12] with some custom source code. For easiness of implementation, the one on top of ABC can only deal with two-input AND gates with inverters. So all the benchmark circuits are first converted into that form and then the proposed method is applied. This implementation applies transformations to all AND gates in a given circuit. The second implementation can deal with arbitrary combinational circuits, but due to its implementation efficiency, a subset of gates is used as a target of transformations.

We have conducted a couple of experiments to see how large the complete sets of test patterns with ISCAS85 and ISCAS89 circuits are. These are the experiments originally conducted in [17] and [19]. For ISCAS89 circuits, only their combinational parts are extracted and processed here. That is, a one time frame of the sequential circuits is processed. As discussed before, the proposed method only makes sense if the complete sets of test patterns, or in order words, the numbers of iterations in the procedure shown above, are not large.

For ease of the experiments, gates are the targets for transformations rather than subcircuits for all the experiments shown in the paper, although the experiments with transformations of subcircuits can be similarly performed.

The first experiment is to apply the proposed method to the entire given circuit, i.e., ISCAS89 circuits. In this case, all the gates in the given circuit are the target of transformations. The experiment is performed with the implementation on top of ABC so all the circuits are first transformed into the ones containing AND gates and inverters.

Two types of transformations are tried. The first one is to only change the polarity of the output of each gate, and the second one is to change the function of each gate to all possible logic functions with the same set of inputs.

In the first case, the number of transformations for each gate p is 2. So if there are m number of gates in the circuit, there are 2^m multiple transformations in total. The results are shown in Table 1. Although all ISCAS89 circuits were tried, small circuits that are trivial and large and take more than 10 h are skipped in the table.

The table shows the statistics for ISCAS89 circuits, including the numbers of AND gates after the original circuits are converted into AND gates and inverters in ABC. "Tests" show the numbers of input patterns generated. The columns of "Multiple trans" show the results when we generate input patterns for all combinations of multiple transformations, and the columns of "Single trans" show the results when we generate input patterns for all single transformations. As expected, it takes much more time under multiple transformations. When it comes to the number of input patterns for complete verification, multiple transformations need several more patterns than single transformations. The last two columns of "Single → Multiple" show the results when we first generate input patterns for all single transformations, and then the additional input patterns are generated for all combinations of multiple transformations. That is, the input patterns are incrementally generated, first, for single transformations, followed by patterns for multiple transformations. As can be seen from the table, this incremental generation works well and larger circuits can be processed.

The numbers of input patterns for a complete verification are in hundreds. This is quite small if we compare it with $2^{|\text{inputs}|}$, and confirms that the proposed approach makes sense in practice.

Table 2 shows the results for the second set of the experiments, i.e., transformations to any logic functions

TABLE 1 Complete Test Generation Results for ISCAS89 Circuits With Transformations of Polarities of Outputs of Gates

ISCAS89	Primary inputs	Primary outputs	FF	ANDs	Multiple trans		Single trans		Single -> Multiple	
					Tests	Time (sec)	Tests	Time (sec)	Tests	Time (sec)
s5378	214	49	180	1,926	294	10,128	118	39	295	1,055
s9234	247	39	212	2,591	>380	>10h	124	118	487	18,746
s13207	700	152	639	4,633	414	22,321	122	323	458	6,712
s15850	611	150	535	5,162	>64	>10h	163	519	489	16,122
s35932	1,763	320	1,729	17,132	>216	>10h	143	4,990	>298	>10h

TABLE 2 Complete Test Generation Results for ISCAS89 Circuits With Transformations to All Possible Logic Functions

ISCAS89	Primary inputs	Primary outputs	FF	ANDs	Multiple trans		Single trans		Single -> Multiple	
					Tests	Time (sec)	Tests	Time (sec)	Tests	Time (sec)
s208	19	2	9	94	>93	>10h	88	2	117	8,615
s298	17	6	15	144	89	2,103.63	105	11	118	874
s344	24	11	16	150	76	44.18	88	3	100	46
s349	24	11	16	154	64	36.52	92	5	102	75
s382	24	6	22	203	111	4,230.53	147	18	159	649
s386	13	7	7	184	>81	>10h	140	12	>148	>10h
s400	24	6	22	211	102	1,779.81	120	16	134	480
s420	35	2	17	188	>78	>10h	142	24	>163	>10h
s444	24	6	22	218	90	1,696.58	114	18	129	1,105
s510	25	7	7	231	>34	>10h	121	30	>82	>10h
s526	24	6	22	266	>49	>10h	146	57	173	3,998
s641	54	24	20	203	264	7,701.97	205	16	233	4,119
s713	54	23	20	217	>249	>10h	198	16	211	1,044
s820	23	19	6	360	>100	>10h	235	146	>114	>10h

with the same set of inputs. Here we use the ABC tool and all internal gates are two-input AND gates. Therefore, the number of all possible logic function is 16. So if there are m gates in the circuit, there are 16^m multiple transformations in total.

As there are many more transformations possible, only smaller circuits can be processed with 10 h of timeout. Circuits larger than the ones shown in the table have timed out. Although the circuits that can be processed are much smaller than the transformations on polarity, with incremental generation, larger circuits can be processed. Also, the numbers of input patterns for a complete verification are 100 or 200, which is, again, much smaller than 2^{inputs}.

Although our formulation restricts the new test patterns to target the remaining cases which cannot be detected by the previous test patterns, the new patterns may also detect the cases which are already detected by the previous test patterns. In that sense, the sets of generated test patterns are not at all nonredundant. They can be very redundant, and so the numbers of test patterns for single trans can be larger than the ones for multiple trans. There may be good heuristics to avoid these situations as much as possible, which we are working on right now, but that is not well matured to be published yet.

Table 3 shows the results for large circuits. The target circuit is OpenRISC, which has 17 701 gates in its combinational part. The entire circuit has 397 primary inputs, 394 primary outputs, and 1860 flipflops. In the experiments, the circuit is a time frame expanded two, three, and four times in order to create larger circuits. The resulting numbers of gates are shown in the table. The total numbers of inputs and outputs for the combinational parts are also shown in the table. In this experiment, our second implementation is used, which deals with the gates of different types and with a different number of inputs.

TABLE 3 Complete Test Generation Results for OpenCore With Transformations to All Possible Logic Functions

Time-frames expanded	Gates	Total in	Total out	LUT replaced	Solved in 5 hours	Average time for solved (s)	Average iterations for solved
2	35,402	4,107	4,114	10	20/20	31	5.5
				20	20/20	62	9.0
				50	20/20	188	19.2
				100	20/20	499	31.7
3	53,103	5,967	5,974	10	20/20	145	11.5
				20	20/20	387	21.0
				50	20/20	1,714	48.1
				100	42358	10,254	84.5
4	70,804	7,827	7,834	10	20/20	1,310	16.9
				20	20/20	1,945	35.0
				50	42024	17,588	64.0
				100	0/20	> 5 hours	N.A.

Here we apply the LUT-based transformations, which allow each gate to have all possible logic functions.

As can be seen from Table II, it is practically impossible to allow all gates to be transformed to any logic function for large circuits. We randomly pick up 10, 20, 50, and 100 gates to be transformed. We tried each case 20 times. For the largest case which has a four time frame expansion, all of them failed to solve within 5 h. For smaller cases, however, most can be solved and the number of input patterns for a complete verification is small, i.e., less than 100.

As can be seen from these experiments, the number of input patterns for a complete verification under the defined multiple transformations is typically quite small, which shows the practical usefulness of the proposed method for logic synthesis, verification, testing, and also debugging. We have done some preliminary work in this direction. In [16], automatic test pattern generation (ATPG) methods for multiple functional faults, where a gate or a subcircuit gate can change its logic functions to any with the same set of inputs, are developed, and it has been confirmed that the numbers of the required test patterns for multiple functional faults are very small compared to the size of the primary input space. In [17], partial logic synthesis, which tries to identify appropriate functions for a set of missing gates or subcircuits for the correct behaviors of the entire circuits, is developed, and it has been confirmed that small numbers of test patterns are sufficient to identify those functions. That is, by checking only small numbers of appropriate test patterns, the functions to be realized can be determined. In [18], ATPG methods, based on the proposed approach targeting complete multiple stuck-at faults, are developed. In spite of the fact that there are exponentially many multiple fault combinations, e.g., for a circuit having 10 000 of two-input gates, there are in total $3^{10\,000\times 3}1 \simeq 10^{14\,313}$ multiple fault combinations, the numbers of test patterns for all nonredundant multiple stuck-at faults are in the order of hundreds or thousands for circuits that have around 10 000 gates.

V. CONCLUSION AND FUTURE PERSPECTIVE

We have presented a method by which logic synthesis and formal verification can be achieved with a small number of input patterns, if all possible circuit transformations are predetermined. The experiments show that even if the number of all combinations of multiple transformations is large (actually, they are exponentially large), the required number of input pattens for a complete analysis is very small, such as hundreds for circuits with hundreds of inputs.

There can be a variety of applications of the proposed method in logic synthesis, formal verification, testing, debugging, and automatic generation of assertions. In all these applications, the complete sets of input patterns generated by the proposed method can be used, instead of all possible input patterns. ■

REFERENCES

[1] A. Mishchenko and R. K. Brayton, "Simplification of non-deterministic multi-valued networks," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2012, pp. 557–562.

[2] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung, "Interpolating functions from large boolean relations," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2009, pp. 779–784.

[3] C.-F. Lai, J.-H. R. Jiang, and K.-H. Wang, "BooM: A decision procedure for boolean matching with abstraction and dynamic learning," in *Proc. Design Autom. Conf.*, Jun. 2010, pp. 499–504.

[4] H.-P. Lin, J.-H. Jiang, and R.-R. Lee, "To SAT or not to SAT: Ashenhurst decomposition in a large scale," in *Proc. Design Autom. Conf.*, Nov. 2008, pp. 32–37.

[5] C.-C. Lin, K.-C. Chen, S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng, "Logic synthesis for engineering change," in *Proc. Design Autom. Conf.*, Jun. 1995, pp. 647–652.

[6] S. A. Seshia, "Sciduction: Combining induction, deduction, structure for verification and synthesis," in *Proc. Design Autom. Conf.*, Jun. 2012, pp. 356–365.

[7] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Proc. 8th Conf. Comput. Aided Verificat.*, Jun. 1996, pp. 428–432.

[8] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. 22nd Int. Conf. Comput. Aided Verificat.*, 2010, pp. 24–40.

[9] S. Sinha, A. Mishchenko, and R. K. Brayton, "Topologically constrained logic synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2002, pp. 679–686.

[10] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. Int. Conf. Softw. Eng.*, May 2010, pp. 215–224.

[11] AIGER. [Online]. Available: http://fmv.jku.at/aiger/

[12] A. Biere, "PicoSAT essentials," *J. Satisfiability Boolean Model. Comput.*, vol. 4, pp. 75–97, 2008.

[13] A. Solar-Lezama *et al.*, "Combinatorial sketching for finite programs," in *Proc. 12th Int. Conf. Architect. Support Programm. Lang. Oper. Syst.*, pp. 404–415.

[14] A. Ling, P. Singh, and S. D. Brown, "FPGA logic synthesis using quantified boolean satisfiability," in *Theory and Applications of Satisfiability Testing*, vol. 3569. Berlin, Germany: Springer-Verlag, 2005, pp. 444–450.

[15] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, "Solving QBF with counterexample guided refinement," in *Theory and Applications of Satisfiability Testing—SAT 2012*, vol. 7317. Berlin, Germany: Springer-Verlag, 2012, pp. 114–128.

[16] S. Jo, T. Matsumoto, and M. Fujita, "SAT-based automatic rectification and debugging of combinational circuits with LUT insertions," in *Proc. Asian Test Symp.*, Nov. 2012, pp. 19–24.

[17] M. Fujita, S. Jo, S. Ono, and T. Matsumoto, "Partial synthesis through sampling with and without specification," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2013, pp. 787–794.

[18] M. Fujita and A. Mishchenko, "Efficient SAT-based ATPG techniques for all multiple stuck-at faults," in *Proc. Int. Test Conf.*, Oct. 2014, Paper no. 26.3.

[19] M. Fujita and A. Mishchenko, "Logic synthesis and verification on fixed topology," in *Proc. 22nd Int. Conf. Very Large Scale Integr.*, Oct. 2014, DOI: 10.1109/VLSI-SoC.2014.7004155.

ABOUT THE AUTHOR

Masahiro Fujita received the Ph.D. degree in information engineering from the University of Tokyo, Tokyo, Japan, in 1985, for his work on model checking of hardware designs by using logic programming languages.

In 1985, he joined Fujitsu as a Researcher and started to work on hardware automatic synthesis as well as formal verification methods and tools, including enhancements of BDD/SAT-based techniques. From 1993 to 2000, he was Director at Fujitsu Laboratories of America and headed a hardware formal verification group developing a formal verifier for real-life designs having more than several million gates. The developed tool has been used in production internally at Fujitsu and externally as well. Since March 2000, he has been a Professor at VLSI Design and Education Center, University of Tokyo. He has done innovative work in the areas of hardware verification, synthesis, testing, and software verification, mostly targeting embedded software and web-based programs. He has been involved in a Japanese governmental research project for dependable system designs and has developed a formal verifier for C programs that could be used for both hardware and embedded software designs. The tool is now under evaluation jointly with industry under governmental support. He has authored and coauthored ten books, and has more than 200 publications.

Dr. Fujita has been involved as program and steering committee member in many prestigious conferences on computer-aided design (CAD), very large scale integration (VLSI) designs, software engineering, and more. His current research interests include synthesis and verification in system on chip (SoC), hardware/software codesigns targeting embedded systems, digital/analog codesigns, and formal analysis, verification, and synthesis of web-based programs and embedded programs.