Reachability Analysis of Sequential Circuits

Jung-Tai Tsai Chun-Yao Wang Kuang-Jung Chang
Department of Computer Science
National Tsing Hua University, HsinChu, Taiwan
lttsai24@gmail.com, wcyao@cs.nthu.edu.tw, jalem1984@gmail.com

ABSTRACT

Reachability analysis is a fundamental technique in the synthesis, verification of VLSI circuits. This paper presents a novel semi-formal approach which combines the advantages of simulation and formal methods to traverse the state space of the FSMs. We conduct the experiments on a set of ISCAS'89 benchmarks. Compared with a previous work which relies on biased random technique, our approach reaches more states with less CPU time.

I. INTRODUCTION

Design verification has become the bottleneck of modern VLSI designs. By industry statistics, 70% of the design efforts are consumed in verification. Without complete verification, erroneous designs may eventually cause enormous economical loss. The infamous Pentium floating-point division bug is such an example. The mainstream of most verification techniques is still simulation-based. This is because the simulation-based verification is easy to apply and can efficiently detect easy-to-detect bugs. However, this common strategy usually cannot effectively target on corner-case bugs. Also, under the tremendous time-to-market pressure, it cannot guarantee bug-free designs.

On the other hand, formal verification uses logic and mathematic methods to prove the correctness of a design. Over the last decade, a wide spectrum of formal verification methods has been proposed. In general, most of them can be broadly classified into two categories: property checking and equivalence checking. Since sequential circuits are typically modeled as finite state machines (FSMs), the first step to perform property checking or equivalence checking on sequential circuits is to compute the reachable states of FSMs. FSM traversal is the process that visits these reachable states of FSMs. To accomplish the FSM traversal, we can use the state transition graph (STG) to explicitly enumerate the states of machines. We can also perform implicit state enumeration without constructing STG to achieve it. Rechability analysis is the task which relies on FSM traversal to explore the state space of FSMs.

Symbolic FSM traversal is a typical approach in FSM traversal [8], [12]. In this symbolic approach, Binary Decision Diagram (BDD) [8], [12] is used to represent sets of states and machine's transition relations. The process of next-state or image computation relies on constructing the BDD of transition relations and operating the existential quantification to quantify out the redundant BDD variables. Additionally, performing computation of reachable states is through iterative symbolic image computation, and the state space is implicitly traversed. When the sets of the reachable states in two consecutive iterations are identical, it reaches the fixed point, which indicates all reachable states have been visited.

In this paper, we propose a semi-formal approach to traverse the state space of FSMs. It combines the advantages of simulation and formal methods. In each iteration, we first randomize a large amount of patterns for parallel simulation. This step efficiently traverses the subset of state space. Then, we use logic implication and circuit structure analysis techniques to determine the reachability of the undecided states in the remainder of the state space. Note that here we also use BDD representation, but just for recording the sets of reached states. The algorithm continues until no newly reached state. The experimental results compared with a simulation-based previous work, which relies on biased random technique [9], show that our approach efficiently reaches more states.

The remainder of this paper is organized as follows. Section II discusses the background of our approach. Section III introduces the proposed approach for reachability analysis. Section IV shows the experimental results. Section V concludes this paper.

II. BACKGROUND

This section first reviews the BDD representation of circuit states. Next, we briefly describe the fundamental element called *search frontier* in our algorithm. *Random pattern generator* (RPG) architecture is also introduced. Finally, we discuss the measurement of controllability of a gate, which will be a guidance in the backward logic implication. Here, we assume each circuit is decomposed into AND, OR, NOT gates, and flip-flops.

A. State Set Representation

1) Representation of Circuit States: Unlike the BDD-based symbolic approach introduced in Section I, we only exploit BDD to record the sets of circuit states. S is generally the set of all flip-flop nodes in the synchronous sequential circuits. There are m flip-flops in the circuit and the state variables in $S = \{s_1, s_2, \ldots, s_m\}$. Given a set of logic value assignments to the corresponding state variables, we can derive a boolean formula which is true for this set of logic value assignments.

Example 2.1: Given $S = \{s_1, s_2, s_3\}$, and let $s_1 = 0$, $s_2 = 1$, $s_3 = 0$, we can derive the boolean formula $\neg s_1 \wedge s_2 \wedge \neg s_3$ for this state, and then we can use BDD to represent this formula.

2) Global BDD and Local BDD: We use the *Global BDD* and *Local BDD* to record the sets of states. The Global BDD can be regarded as a global variable $global\_bdd(t)$ to record all the reachable states from the timeframe 0 to t. The Local BDD is a local variable $local\_bdd(t)$, which is used to represent the reached states at the exact timeframe t.

```
Reachability()
{
  t = 0;
  global_bdd(t) = S0; // initial state
  repeat
    t = t + 1;
    global_bdd(t) = global_bdd(t-1) ∪ new_states(local_bdd(t-1));
  until global_bdd(t) = global_bdd(t-1);
}
```

Fig. 1. The pseudo code of combining Global BDD and Local BDD.

Fig. 1 shows the pseudo code of operations that combines the Global BDD and the Local BDD. The function $new\_states()$ extracts the disjoint cubes from the Local BDD. It also verifies if the cubes of $local\_bdd(t)$ are the subsets of $global\_bdd(t)$. If not, that means new states are reached, then we add these cubes into $global\_bdd(t)$ using the *union* operation. The iterative process terminates when the Global BDDs in the two successive iterations are identical, i.e., $global\_bdd(t) = global\_bdd(t-1)$.

B. Search Frontier

In Section II.*A*, we perceive that the cube which is not the subset of the BDD may be useful to visit new states. Such a cube is called a *search frontier*. A given initial state S_0 is also a search frontier. In the beginning, the Global BDD is empty, so state S_0 is not the subset of Global BDD. Hence, S_0 is the only search frontier to visit new states at the timeframe 0. That is, the reachable states at the timeframe 0, which are represented as $local\_bdd(0)$, are visited only via S_0.

978-1-4244-5271-2/10/$26.00 ©2010 IEEE

In general, after extracting some cubes from $local\_bdd(t-1)$, and verifying them not a subset of $global\_bdd(t-1)$, we can determine the search frontiers at this timeframe. We denote the i^{th} search frontier at timeframe t as $s\text{-}frontier(i,t)$, and the Local BDD at the timeframe t, $local\_bdd(t)$ can be computed as

$$local\_bdd(t) = \bigcup_{i=1}^{n} reach(s\text{-}frontier(i,t)) \quad (1)$$

where n is the number of the search frontiers. The function $reach()$ computes the reachable states via the $s\text{-}frontier(i,t)$, and the union of these reachable states can be considered as the next state set in the timeframe t.

C. Random Pattern Generator (RPG)

The RPG architecture is shown in Fig. 2. It comprises two components, one is random pattern generator (RPG) itself and the other is the circuit under test (CUT) SF. SF actually is the *combinational* part of a sequential circuit and we set fixed logic value '0' or '1' to the pseudo-primary inputs (PPIs) based on a search frontier from the previous timeframe. Assume SF is an N-input M-output network. Then

$$N = I + R - R_f \quad (2)$$

where I is the number of primary inputs (PIs), R is the number of PPIs, and R_f is the number of PPIs associated with logic values. M refers to the number of PPOs in the circuit. When N is determined, the RPG produces N outputs and assigns them to the inputs of SF. The parameter r in the RPG indicates that the RPG can simultaneously generate 2^r parallel patterns with the support of GMP library [13]. In addition, if a PPI has set a fixed logic value, it will be extended to 2^r bits in length. These parallel patterns are used for simulation for reaching new states. This process can be seen in detail in Section III.

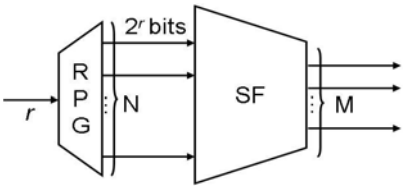

Fig. 2. The architecture of RPG.

D. Controllability

Controllability of a node in a circuit is a measurement to show the difficulty to set a value to this node from the PIs. As mentioned in Section II.C, following the parallel pattern simulation, each node n of network is associated with a 2^r-bit set, $n\text{-}Bitset$. $|n\text{-}Bitset|$ represents the number of 1's in this $n\text{-}Bitset$. We denote the 1's controllability and 0's controllability of the node n as $C_1(n)$ and $C_0(n)$, respectively. Based on this simulation, $C_1(n)$ and $C_0(n)$ can be approximated as Equation (3) and Equation (4)

$$C_1(n) \approx \frac{|n\text{-}Bitset|}{2^r} \quad (3)$$

$$C_0(n) \approx 1 - C_1(n) \quad (4)$$

We use this measurement to guide the decision in the process of logic implication in our approach. It will be discussed in detail in Section III.

III. Our Approach

In this section, we integrate some preliminaries mentioned in Section II to complete our approach. Our algorithm comprises two main components. The *Parallel Random Pattern Simulation* serves as the first stage to efficiently traverse the partial state space via a specific search frontier. Then, we perform *Backward Justification* to determine the reachability of those undecided states in the remaining state space via the same search frontier. The details of these two stages will be described in the following subsections.

A. Parallel Random Pattern Simulation

As mentioned in Section II.C, for a search frontier, the RPG produces N outputs and assigns them to the N-input SF. After parallel pattern simulation, each node n is associated with a 2^r bit set, $n\text{-}Bitset$. We denote the i^{th} bit of $n\text{-}Bitset$ as $n\text{-}Bitset(i)$, and the bit index always starts with 1 and it counts from the right-most bit to the left-most bit. The corresponding logic value in the $n\text{-}Bitset(i)$ is obtained by simulating the i^{th} random pattern.

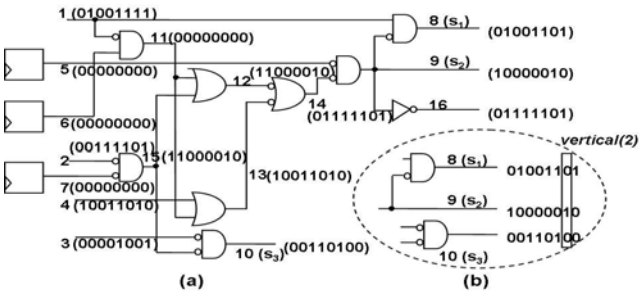

Fig. 3. An example to illustrate parallel random patterns simulation.

Example 3.1: Fig. 3(a) shows an ISCAS'89 circuit s27, nodes $1 \sim 4$ in the circuit are PIs, nodes $5 \sim 7$ are PPIs, nodes $8 \sim 10$ are PPOs, and node 16 is a PO. Given the initial state $S_0 = \{0,0,0\}$, consider $(0,0,0)$ as the only search frontier $s\text{-}frontier(1,0)$ to visit new states at the timeframe 0. Thus, $(0,0,0)$ are assigned to the PPI nodes 5, 6, and 7, respectively. Assume the parameter r of RPG is set to 3, and based on Equation (2) we can get $N = 4$ ($I = 4, R = 3, R_f = 3$). Then 2^3-bit vectors are generated and assigned to the SF with 4 inputs. The logic values of nodes $5 \sim 7$ are also extended to 2^3 bits. Finally, we simultaneously simulate them.

We create boolean variable $s_1 \sim s_3$ for each flip-flop in Fig. 3(a), and in what follows, we use nodes $s_1 \sim s_3$ to substitute nodes $8 \sim 10$ for simplicity. After parallel pattern simulation, we can get $s_1\text{-}Bitset$ = (01001101), $s_2\text{-}Bitset$ = (10000010), and $s_3\text{-}Bitset$ = (00110100). In Fig. 3(b), the i^{th} output vector evaluated by simulating i^{th} random pattern with the initial state $(0,0,0)$ is denoted as $vertical(i)$, and

$$vertical(i) = (s_1\text{-}Bitset(i), s_2\text{-}Bitset(i), \ldots, s_m\text{-}Bitset(i)) \quad (5)$$

By random simulation, we can efficiently get different $vertical(i)$ and regard them as reachable states.

Example 3.2: Fig. 3(b) shows that $vertical(2) = (s_1\text{-}Bitset(2), s_2\text{-}Bitset(2), s_3\text{-}Bitset(2)) = (0,1,0)$, and there are four different $vertical(i)$, $(1,0,0)$, $(0,1,0)$, $(1,0,1)$, and $(0,0,1)$, which are regarded as four different reached states. Fig. 4(a) represents these states using BDD.

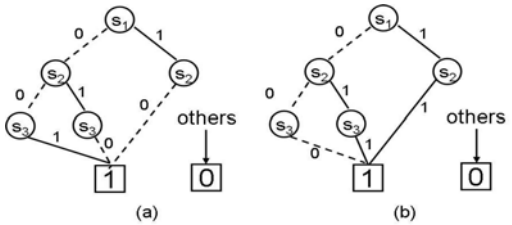

Fig. 4. BDD representation of reached states and undecided states.

B. Backward Justification

Following Section III. *A*, partial reachable states have been visited via the S_0, $s\text{-}frontier(1,0)$. Then we determine the reachability of other undecided states via the same search frontier. Fig. 4(b) shows the remaining state space of Fig. 4(a). We use depth-first-search (DFS) manner to traverse this BDD, and extract the cubes in it. Note that each cube is disjointly extracted. We denote the cube which represents the undecided states as a $u\text{-}cube$ and denote the i^{th} cube as $u\text{-}cube(i)$.

In the process of backward justification, the measurement of controllability serves as a guidance to determine the justification ordering in PPOs and of backtrace paths. The controllability of all nodes can be simultaneously computed based on parallel random pattern simulation in the first stage. According to the approximate equations (3) and (4), we can easily compute $C_1(n)$ and $C_0(n)$ of node n, e.g., $C_1(15) = \frac{3}{8}$, $C_0(15) = \frac{5}{8}$ are computed from Fig. 3(a).

There are three components of the backward justification as follows: 1) *Forward implication :* For an arbitrary search frontier $s\text{-}frontier(i,t)$, we have corresponding logic values at PPIs. We directly propagate these values to other nodes as possible as we can. Therefore, some nodes also have the fixed logic values, and these values logically set restrictions when we perform the backtracing.

2) *Justification ordering :* After forward implication, we set the logic values of PPOs based on an undecided cube, $u\text{-}cube(i)$, which is the i^{th} cube extracted from the remainder of state space. Then we perform backtracing. But before that, we have to determine the ordering of justification among all PPOs. Different orderings may result in different effects on efficiency. Fig. 5 illustrates the effect of this ordering issue. The justification ordering of Fig. 5(a) is $(s_1 \rightarrow s_2 \rightarrow s_3)$, and a conflict occurs after backtracing from the last PPO, s_3. A conflict represents the corresponding undecided states are not reachable. But if we apply another ordering $(s_1 \rightarrow s_3 \rightarrow s_2)$ in Fig. 5(b), this conflict occurs earlier. An earlier occurred conflict can improve the efficiency of backward justification. Thus, we propose a heuristic that uses the controllability of PPO as a guidance to determine the justification ordering. As we know, controllability is the probability of a signal value at a node being set to 0 or 1. If a node with lower probability (controllability) of being its set value, we consider that this node with its set value are difficultly justified at PIs, and a conflict may easily occur in the backtracing. Thus, our heuristic relies on this concept.

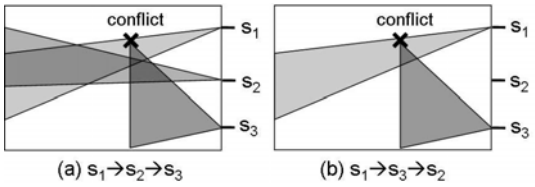

Fig. 5. Justification ordering.

3) *Controllability as a guidance of backtrace :* We also heuristicly use controllability as a guidance in the selection of backtrace paths. There are two selection criteria. For the gate that requires all inputs having its specific values, we attack the hardest one among all inputs. For example, Fig. 6(a) shows node 14 in Fig. 3(a). Assume node $14 = 0$ is in the backtrace path, it requires both node $12 = 1$ and node $13 = 1$. Based on the controllability of node 12 and 13, $C_1(12) = \frac{3}{8} < C_1(13) = \frac{4}{8}$, we choose the harder one (lower controllability), i.e., $(14 \rightarrow 12)$, for backtracing. For the gate that only requires certain input having its specific value, we select the easiest one among all inputs. For example, Fig. 6(b) shows node 10 in Fig. 3(a). Assume node $10 = 0$ is in the backtrace path, it requires either node $3 = 1$ or node $15 = 1$. Thus, based on the controllability of node 3 and 15, $C_1(15) = \frac{3}{8} > C_1(3) = \frac{2}{8}$, we choose the easiest one (higher controllability), i.e., $(10 \rightarrow 15)$, for backtracing.

To reduce the complexity of the backward justification, we just apply some simple but effective strategies as mentioned into this process. Thus, we may not completely reach all reachable states at a timeframe. Nevertheless, our approach never reaches any unreachable states.

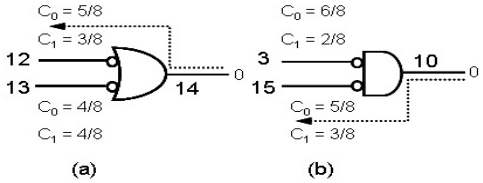

Fig. 6. Guidance of Backtrace.

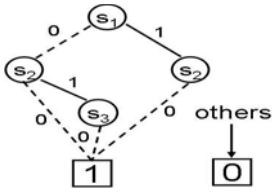

Fig. 7. Reachable state set at the timeframe 0.

In summary, we use parallel random pattern simulation as the first stage to efficiently traverse four states, $(1,0,0)$, $(0,1,0)$, $(1,0,1)$ and $(0,0,1)$ via the initial state $S_0 = \{0,0,0\}$. Then perform Backward Justification to reach additional state $(0,0,0)$ from the remaining state space. According to Equation (1), we can compute $local\_bdd(0) = reach(s\text{-}frontier(1,0))$ as shown in Fig. 7, which represents the reachable state set at the timeframe 0. In the timeframe 1, we also extract $s\text{-}frontier(1,1) = (0,0,X)$, $s\text{-}frontier(2,1) = (0,1,0)$, and $s\text{-}frontier(3,1) = (1,0,X)$ from $local\_bdd(0)$ as shown in Fig. 7 (from left to right), and then use each search frontier as the state value to explore the reachable state space. The iterative process eventually terminates when Global BDDs in the two successive iterations are identical, i.e., $global\_bdd(t) = global\_bdd(t-1)$.

IV. Experimental Results

We conduct the experiments on a set of ISCAS'89 benchmarks within SIS [11] environment. Each benchmark is decomposed into AND, OR, NOT gates, and flip-flops for simplicity. We then take a previous work [9] which relies on a biased random technique for comparison.

The basic idea of this previous work is to iteratively derive a set of input probabilities used for random simulation to explore the next state space based on a current state. The objective is to derive the input probabilities such that each controllable state variable has probability of 0.5, i.e., the same probability of being 1 or 0. It expects that this probability assignment can maximize the next state exploration.

Both [9] and our approach are implemented on a 1280 MHz Sun Blade 2500 workstation with 4 GBytes memory. We set $r = 10$ in our parallel random pattern simulation. Table I shows the comparison about the CPU time and reached states between [9] and our approaches.

In Table I, the first three columns show the name of each circuit, the number of inputs, and the number of flip-flops, respectively. We repeat our algorithm 10 times and take an average of these 10 trials. Columns six and seven show the average number of reachable states, and the average required CPU time in our approach. Then, we spend the same amount of CPU time in our approach to run the algorithm in [9], and record the number of reached states as shown in column four. Column five shows the run time of [9] when it reaches the same number of states as ours. Column eight shows the ratio of reached states between ours and [9] in the same amount of CPU time. The last column shows the ratio of the CPU time between [9] and ours for reaching the same amount of states.

Take s344 as an example, after running 299.29 seconds on average, our approach terminates and reaches 2208.6 states while [9] only reaches 1524 states. Thus, the state ratio is 1.45. Also, [9] spends 6907.34 seconds to reach 2208.6 states while our approach only needs 299.29 seconds. Thus, we achieve 23.08 speedup over [9]. The experimental results show that on average we visit 19% more reachable states than [9], and achieve 7.04 speedup.

TABLE I
THE COMPARISON OF THE CPU TIME AND THE REACHED STATES BETWEEN [9] AND OUR COMPLETE APPROACH.

Circuits	|PI|	|FF|	[9]	[9]	Ours		State ratio	Time ratio
			State	Time (sec.)	State	Time (sec.)	Ours / [9]	[9] / Ours
s344	**9**	**15**	**1524**	**6907.34**	**2208.6**	**299.29**	**1.45**	**23.08**
s349	9	15	1526	7099.39	2207.2	300.99	1.45	23.59
s382	3	21	8725	2511.86	8865.0	1475.05	1.02	1.70
s400	3	21	8797	2594.82	8865.0	1472.05	1.01	1.76
s444	3	21	8765	2484.94	8865.0	1066.61	1.01	2.33
s526	3	21	8763	3027.60	8868.0	1047.40	1.01	2.89
s641	35	19	1025	1384.46	1533.2	299.49	1.50	4.62
s713	35	19	1109	1476.13	1535.1	309.40	1.38	4.77
s1196	14	18	2597	5082.47	2614.0	1755.36	1.01	2.90
s1238	14	18	2591	5075.53	2613.2	1868.69	1.01	2.72
Average ratio							***1.19***	***7.04***

In Table II, we remove the backward justification process from our original approach, and repeat this trial 10 times and take an average of these 10 trials. Then, we compare it with our original complete approach. Columns four and five show the number of reachable states, and the required CPU time in our approach without backward justification. The data with backward justification are shown in columns six and seven. These data are the same with that in Table I. Column eight shows the ratio of reachable states between w/ and w/o backward justification in our approach. The last column shows the ratio of the CPU time between w/o and w/ backward justification in our approach. Next, we analyze the results of these circuits. For s344 and s349, we can find that the number of reached states decreases and the required CPU time increases when the backward justification process is removed. That means the process of backward justification may reach some reachable states which are hard to be reached by random simulation.

TABLE II
THE COMPARISON BETWEEN WITHOUT AND WITH BACKWARD JUSTIFICATION IN THE ALGORITHM.

Circuits	|PI|	|FF|	w/o		w/		State ratio	Time ratio
			State	Time (sec.)	State	Time (sec.)	w/ / w/o	w/o / w/
s344	**9**	**15**	**2130.6**	**338.45**	**2208.6**	**299.29**	**1.04**	**1.13**
s349	**9**	**15**	**2123.1**	**332.87**	**2207.2**	**300.99**	**1.04**	**1.11**
s382	3	21	8865.0	1468.78	8865.0	1475.05	1.00	1.00
s400	3	21	8864.4	1468.39	8865.0	1472.05	1.00	1.00
s444	3	21	8865.0	1065.54	8865.0	1066.61	1.00	1.00
s526	3	21	8868.0	1037.26	8868.0	1047.40	1.00	0.99
s641	35	19	1534.6	257.00	1533.2	299.49	1.00	0.86
s713	35	19	1536.4	257.74	1535.1	309.40	1.00	0.83
s1196	14	18	2613.3	510.91	2614.0	1755.36	1.00	0.29
s1238	14	18	2613.1	509.40	2613.2	1868.69	1.00	0.27
Average ratio							***1.01***	***0.85***

TABLE III
THE COMPARISON BETWEEN WITHOUT AND WITH BACKWARD JUSTIFICATION FOR s344.

Timeframe t	w/o		w/	
	Time (sec.)	State	Time (sec.)	State
0	**0.41**	**437**	**0.45**	**513**
1	**53.23**	**943**	**1.23**	**995**
2	79.30	1303	23.82	1378
3	138.47	1563	80.57	1751
4	215.93	1991	155.81	2114
5	304.26	2118	253.84	2253
6	343.27	2118	297.61	2253

In a more detailed analysis to s344, we also show the increase of state number in each timeframe for these two approaches in Table III. The first column is the timeframe index. s344 reaches its fixed point at the timeframe 6. Column two shows the CPU time at the end of each timeframe, and column three shows the total number of reached states. Columns four and five are similar to columns two and three, respectively, but just for the complete approach. For example, for our approach w/o backward justification, we can totally reach 437 states at the end of timeframe 0 with 0.41 seconds CPU time. But for the with backward justification version, the number of reached states is 513 in 0.45 seconds. These results mean via $s\text{-}frontier(1, 0)$, the approach w/ backward justification reaches 76 more states than that of w/o, and the backward justification is useful to maximize the next state exploration.

But for the other circuits in Table II, we find that each circuit of them has a similar image in the corresponding timeframe by using both approaches. That means the backward justification does not reach additional states in these circuits. This is because for the circuits with certain functionality, e.g., counter, the reachable states at each timeframe are easy to reach by random simulation. Although the backward justification in our complete approach does not always contribute new reached states, and may required more CPU time, we still think this process is important and is included in our algorithm. This is because the results of random simulation are unpredictable. We add the backward justification process to constantly compensate the results of the random simulation. In addition, reaching more states is more important to verification. Thus, although spending more CPU time, the backward justification process still deserves to be maintained in our approach.

V. CONCLUSIONS

Reachability analysis is desired for VLSI circuit synthesis and verification. This paper presents a novel semi-formal approach which combines the advantages of simulation and formal methods to traverse the state space of the FSMs. A large amount of parallel random pattern simulation can efficiently traverse partial state space, and formal methods serves as the important role to compensate the insufficiency of the first stage. The experimental results show that on average our complete algorithm obtains 19% more reachable states than previous work [9] and achieves 7.04 speedup.

REFERENCES

[1] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 4, pp. 401-424, Apr. 1994.

[2] J. R. Burch, E. M. Clarke, and D. E. Long, "Representing circuits more efficiently in symbolic model checking," in *Proc. 28th Design Automation Conf.*, pp. 403-407, 1991.

[3] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *Proc. Int. Conf. Very Large Scale Integration*, pp. 49-58, 1991.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *Proc. 27th Design Automation Conf.*, pp. 46-51, 1990.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proc. Fifth Ann. IEEE Symp. Logic in Computer Sci.*, pp. 428-439, 1990.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Infor. Computation*, vol. 98, no. 2, pp. 142-170, June 1992.

[7] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Int. Workshop on Automatic Verification Methods for Finite State Systems*, pp. 365-373, June 1989.

[8] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *Proc. Int. Conf. Computer-Aided Design*, pp. 126-129, Nov. 1990.

[9] Y.-M. Kuo, C.-H. Lin, C.-Y. Wang, S.-C. Chang, and P.-H. Ho, "Intelligent random vector generator based on probability analysis of circuit structure," in *Proc. of Int. Symp. Quality Electronic Design*, pp. 344-349, 2007.

[10] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz, "Structural FSM Traversal," *IEEE Trans. Computer-Aided Design*, vol. 23, no. 5, pp. 598-619, May 2004.

[11] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[12] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *Proc. Int. Conf. Computer-Aided Design*, pp. 130-133, Nov. 1990.

[13] http://www.swox.com/gmp