Effective Safety Property Checking Using Simulation-Based Sequential ATPG

Shuo Sheng  
Department of ECE  
Rutgers University  
Piscataway, NJ, 08854  
shuo@ece.rutgers.edu

Koichiro Takayama  
Fujitsu Labs. of America Inc.  
555 Lawrence Expressway  
Sunnyvale, CA, 94086  
ktakayam@fla.fujitsu.com

Michael S. Hsiao  
Bradley Department of ECE  
Virginia Tech.  
Blacksburg, VA, 24061  
mhsiao@vt.edu

ABSTRACT

In this paper, we present a successful application of a simulation-based sequential Automatic Test Pattern Generation (ATPG) for safety property verification, with the target on verifying safety property of large, industrial-strength, hardware designs for which current formal methods fail. Several techniques are developed to increase the effectiveness and efficiency during state exploration and justification of the test generator for verification, including (1) incorporation of a small combinational ATPG engine, (2) reset signal masking, (3) threshold-value simulation, and (4) weighted Hamming distance. Experimental results on both ISCAS89 benchmark circuits and real industry circuits have shown that this simulation-based verifier achieves better or comparable results to current state-of-the-art formal verification tools BINGO and CHAFF.

Categories and Subject Descriptors
B.7.2 [INTEGRATED CIRCUITS]: Design Aids, Verification, Simulation; B.8.1 [PERFORMANCE AND RELIABILITY]: Testing

General Terms
Algorithms, Verification

Keywords
Verification, Simulation-Based, Sequential ATPG

1. INTRODUCTION

The ever-increasing complexity and circuit size have made verification and testing bottlenecks in the VLSI design cycle. In this paper, we address the problem of safety property verification in model checking and propose a simulation-based solution. The safety property verification problem involves checking if a property is held or violated in the design, and the current solutions essentially involve extensive state traversal in the finite-state-machine (FSM) of the design. The FSM traversal problem targetting property verification can be further decomposed into three subproblems: (1) how to effectively represent the states; (2) how to effectively represent the state transition function; (3) how to effectively traverse the state space. The main difficulty in all three problems lies in the exponential gap between design size and the number of states to explore.

Verification based on functional simulation is a very popular approach due to its simplicity. Its effectiveness, however, largely depends on the quality of vectors that are simulated. For large designs (which often implies large state space), exhaustive simulation would not be feasible, and some corner-case bugs can often be missed. On the other hand, significant advances in formal methods to design verification have been published. These formal methods can rigorously prove if the (small to medium-sized) design is correct.

Model checking and equivalence checking are two verification problems for which successful formal methods have shown promise. Model checking determines whether or not the design satisfies a given set of properties. If a property is proven to be violated, a counterexample (or error trace) can be generated. Equivalence-checking, on the other hand, attempts to rigorously prove if the implementation circuit is functionally equivalent to the specification circuit (golden model). Formal methods for both model-checking and equivalence-checking frequently involve FSM traversal.

In this paper, we target model-checking of safety properties. It is noted, however, that our method may also be applicable to equivalence checking as well.

The current model checking methods fall into three categories: (1) explicit methods, (2) symbolic methods, (3) ATPG or SAT based methods. Explicit methods take the finite state model of the design and enumerate the states explicitly. The example is Murϕ [2]. The limitation of this approach is its lack of scalability to larger design due to the explicit storage of the enumerated states. Symbolic model checking [1] utilizes Reduced Ordered Binary Decision Diagrams (ROBDDs) to compactly represent and manipulate the set of reachable states and the state transition functions. ROBDDs represent the sets of state and transition functions implicitly by its canonical structure. The attractive feature...
of OBDDs is that the OBDD size may be much smaller than the states represented (with a given variable order). Thus, it is able to handle more states than explicit methods. Example verifiers in this class are SMV [3] and BINGO [18]. Though more efficient than explicit methods, the primary limitation of BDD-based approaches is the “memory explosion” problem, which occurs when the BDDs for representing the state transition functions of the circuits may not be constructed entirely [7]. As a result, methods other than BDDs have been proposed, such as SAT [4] and ATPG-based model-checking approaches [8, 13, 15]. The ATPG-based model-checking approach does not explicitly store the complete state space or build a canonical representation of the Boolean state transition function, instead the state transition is done by forward simulation and the state information is learned on the fly. A solution is derived when sufficient knowledge has been acquired. However, ATPG could potentially take a long time when a hard property for a large circuit is encountered. In essence, BDD approaches are limited in space (memory) and could be limited in time if variable ordering is needed, while ATPG approaches are limited in time.

In terms of sequential ATPG, two approaches are possible: deterministic and simulation-based. Currently, almost all previous work on using ATPG for verification are deterministic [8, 13, 15] because they can prove correctness (i.e., when the modeled fault is untestable). In this case, a branch-and-bound algorithm is employed to exhaust the search space progressively so that no input vector could be found to detect the targeted fault; thus, we can safely claim that this property is held as the search space has been exhausted. If the procedure succeeds in finding a sequence to detect the target property, a counter example is generated to invalidate the property. As ATPGs are limited in time, the procedure aborts on difficult properties because of limited time, which is restricted in practice by a given limit in the number of backtracks, time-limit, etc. When the target objective is aborted, we can claim nothing about the property. Although various techniques have been developed to prune the search tree (e.g., given the recent significant progress in Boolean Satisfiability (SAT) solver [5, 6]) so that many unnecessary backtracks are reduced, aborts still occur because the deterministic ATPG intrinsically tries to exhaust the search space. The major difference between SAT and deterministic sequential ATPG are, though both are branch-and-bound, ATPG can target sequential time-frames implicitly, while SAT generally targets it by explicitly unrolling frames.

The other category, simulation-based ATPG for verification, is investigated in this work. Unlike the deterministic counterpart which tries to exhaust the complete search space, simulation-based ATPG does a “random walk” in the state space, hoping to hit the “target” in a limited number of steps. The walk is guided by some heuristics, thus it is not an entirely random traversal. At each step, a number of trial steps are made and evaluated by a cost function which measures how far the state of the circuit is from the target state. The cost function is computed through either logic simulation or fault simulation. The step with the least cost is picked as the next step and the process is repeated until the target is reached or certain limit is exceeded. Such ATPGs have the advantage of scaling to large designs because of its simulation-based nature, where only forward simulation is used and backtracks never occur. However, the major drawbacks are: (1) it can not prove correctness, due to simulation-based ATPG’s inability to identify untestable faults; (2) a good cost function is needed to guide the search, otherwise a lot of time will be wasted on traversing the useless states. Although the simulation-based methods cannot prove correctness in verification, it fits quite well the objective of “debugging”. In practice, for a large industry design, the verification objective is usually concerned less with providing formal proofs of correctness than with finding bugs. In other words, designers are more concerned with finding a counter example to falsify the large design, which is generally easier than proving if it is 100% correct.

Our main contribution in this work is on elimination of the two “drawbacks” of simulation-based ATPG for verification. First, we incorporate a small deterministic engine, which is used to generate justification objective for the simulation engine as well as identify certain number of untestable faults. Second, we developed several techniques to refine the cost function in the simulation engine to efficiently guide the state traversal during the “bug hunt”. Experimental results on both large ISCAS89 benchmark and real industrial circuit show our simulation-based approach achieved comparable or better performance than the existing formal approaches including CHAFF [6].

The organization of the paper is as following: Section 2 present some preliminaries such as problem formulation and introduction of sequential ATPG tool STRATEGATE; the four of our approaches for using sequential ATPG for verification are described in Section 3; Section 4 gives the overall procedures and discusses the “local minima” problem encountered; experimental results are presented in Section 5 and followed by conclusion in Section 6.

2. PRELIMINARIES

The prototype of the sequential ATPG we use for verification is STRATEGATE [9, 10]. It is a simulation based ATPG that uses genetic algorithm (GA) as the core algorithm for test generation. In GA, a population contains a set of individuals or strings. Every string is a candidate solution for the problem at hand. Each is associated with a fitness, which measures the quality of this individual for solving the problem. In test generation context, this fitness measures how good the candidate sequence (bit string) is for exciting or propagating the target faults. The fitness evaluation is done by fault simulation. Based on the evaluated fitness, evolutionary process of selection, crossover and mutation are used to generate a new population from the existing population. The process is repeated until the fitness of the best individual is good enough.

In the application context, we focus on verifying a class of safety properties which are expressed in CTL as $EF(p)$. The property to be verified is converted to a stuck-at-0 fault by constructing a monitor circuit and add it to the original circuit. The output of the monitor becomes one of the primary output (PO) of the transformed circuit, which is shown in Figure 1. Then the problem of verification becomes test generation for this stuck-at-0 fault. Because this is a PO fault, all we need is fault excitation. STRATEGATE performs test generation for this target fault in the following framework: first, a combinational ATPG is performed to derive a test (a vector of primary inputs $I_p$ plus state $S_{target}$) which can excite the fault in one time frame, assuming full control of the state; second, justification process – a search is done for a sequence $\{I_0, I_1, \ldots, I_n\}$ that can justify the
state $S_{\text{target}}$ from a given reachable state. If the justification process succeed, we then obtain a test sequence $\{I_0, I_1, \ldots, I_n, I_P\}$ that can detect the fault, which falsifies the property. If it failed, the only conclusion we can make is that we can not prove this property to be wrong. This step is hard because it involves essentially sequential ATPG.

3. OUR APPROACH TO VERIFICATION

Our work focuses on developing several techniques to build a strong state justification engine into the simulation-based test generator. The techniques we developed are described in the following subsections.

3.1 Incorporation of Deterministic ATPG

The first work is embedding a small combinational deterministic test generator into the simulation-based framework. The original STRATEGATE uses single-time-frame GA to do a simulation-based combinational test generation. However, it can not guarantee a solution even if one exists. Therefore, we put in a PODEM-based deterministic engine to help STRATEGATE. Since this deterministic engine is only called for combinational test generation, its cost is low even for large circuits. We modified the PODEM algorithm so that it can generate multiple solutions (if they exist) for a target fault. This is desirable because each solution is associated with a target state to be justified. When justification fails for one (unreachable) state, we can switch to the next solution. This deterministic engine also lends another advantage: it can identify some untestable faults. Because our deterministic engine is mainly combinational oriented, it is not powerful enough to identify sequentially untestable faults. However, we can identify all combinational untestable faults (given enough backtrack limits) and thus can prove the correctness of some properties.

3.2 Reset Signal Masking

A technique called reset signal masking is incorporated in the GA generation process to avoid revisiting previously visited states (thus forming cycles) during state traversal. Usually, designers will put in reset (or partial reset) signals into circuit for Design for Testability (DFT) purposes. Assertion of these signals will force the circuit state to return to certain reset or partial-reset states, thereby repeating cycles or loops in the FSM traversal. This will significantly reduce the number of new states visited and limit the depth the test generator can reach in the State Transition Graph (STG). The hard properties to be verified usually require a long sequence of states be traversed before the target states are reached. Therefore, we developed the reset signal masking technique in the GA to avoid resetting or partially resetting the state.

Definition 1. Reset Power $O(1)$ for a given PI is the number of FFs that can be reset to a specified value by applying a $O(1)$ at this PI and $X$ at all other PIs, with the initial state of circuit being all $X$.

Reset power measures how powerful a single PI is in resetting the Finite State Machine (FSM) of the circuit. If a PI can reset all the FFs, then its reset power is $n_{FF}$, the total number of flip-flops in the circuit. Reset power for each PI can be identified by parallel logic simulation, starting from the all-unknown state, which takes negligible time. From the reset power, we define the following Reset Mask Probability.

Definition 2. Reset Mask to 1 Probability: if the reset power $O$ of PI$_j$ is greater than 0, then its reset mask to 1 probability is

$$\text{prob}(\text{PI}_j = 1) = \frac{\text{Reset Power}_j \times k}{n_{FF}},$$

where $k$ is a weight used to scale the probability value. Sometimes $n_{FF}$ is far greater than the reset power value, which makes the corresponding reset mask probability near to zero. In this case, we use $k$ to scale the value to a noticeable range. Similarly, we can also define reset mask to 0 probability, which is analogous to its reset mask to 1 probability counterpart. After we identify the reset power, reset mask probability is calculated and applied in GA after crossover and mutation operation.

Table 1 reports the reset signals we identified from the circuits we were using in the experiments. For each circuit, the number of FFs in this circuit is first given. Next, each subsequent column reports the number of PIs that can reset the corresponding number of flip-flops. For example, the value "12" under the column "1452" for circuit s85584 indicates that there is one PI that can reset 1452 FFs for this circuit. This is actually a global reset signal since all FFs are reset. For circuit s35784, the value "4" under column "3" indicates that there exist 4 PIs that can reset groups of 3 FFs. We noticed from the table that some PIs have stronger resetting powers than others, e.g. in the same circuit of s85584, there is another PI that can reset 103 FFs.

3.3 Threshold-Value Simulation

Threshold-value model of logic gates was first proposed by Cheng and Agrawal [11]. They developed a test generator based on threshold value simulation [12]. Threshold value model is an analog model of logic gates which not only keeps the Boolean logic value information but also carries the dynamic controllability information with the corresponding threshold value. A logic ‘0’ is represented by a value within the real value range [0.0,1] and a logic ‘1’ is represented by a value within [0.9,1]. Therefore, it leaves space for the concept of “strong” and “weak” 0 and 1. The closer the threshold value to 0, the stronger the logic ‘0’ is. Conversely, the closer the threshold value to 1, the stronger the logic ‘1’ is. The evaluation of a gate is done by mapping the average input value of the gate to a output value within range [0,1] through the threshold function of the gate. The detail of the threshold function of each gate type is defined in [12]. In implementation, floating value is represented as integer to eliminate floating-point operation.

In our work, we use the threshold-value simulation to refine the GA fitness function for state justification. The
original STRATEGATE uses the number of FFs of the current state that matches the target states as the fitness function. However, it can not distinguish two different sequences where both justify the same subset of FFs but neither of them justifies the entire state. Threshold value simulation, in this case, will help us further distinguishing the two by measuring how strong or weak the 0s or 1s are for those unmatched (failed) FFs. Figure 2 shows a simple example of how threshold value model differentiates two cases which have the same logical value. Suppose we want to justify a FF whose input is a 3-input AND gate. Assume the objective is to justify the FF to be 1. In figure 2(a) and figure 2(b) the objective both failed because a logic ‘0’ is obtained at the output of the AND gate. However, it is observed that the causes of the logic ‘0’ at the AND gate are different in the two cases. In figure 2(a), we have two inputs to the AND gate being ‘1’ and one input being ‘0’. In figure 2(b), we have all three inputs to the AND gate being ‘0’. This implies that (a) has a weaker ‘0’ at the output of AND gate than (b) and hence is more preferable because we just need to flip one of the three inputs to get a 1 to justify the objective FF while for case (b), we need three. This is reflected by a output threshold value the AND gate (obtained through threshold value simulation) being 0.074 for (a) and 0 in case (b). As a result, (a) is favored instead of (b) in GA evolution.

![Figure 2: Two cases in threshold value simulation](image)

With threshold value simulation, we use the following fitness function in GA for state justification

$$fitness = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{|FF_{\text{thresh}}(i) - FF_{\text{obj}}(i)|}$$

where $FF_{\text{thresh}}(i)$ is the threshold value of the $i^{th}$ unjustified FF in the target state, $FF_{\text{obj}}(i)$ is the objective value of that FF (0 or 1, generated by PODEM). $N$ is the total number of unsatisfied objectives. For the justified FFs in the target state, $FF_{\text{thresh}}(i) = FF_{\text{obj}}(i)$, we do not include them into the fitness function because otherwise the denominator will become zero. The fitness is defined in the reciprocal form because GA will maximize it.

3.4 Weighted Hamming Distance

The fourth technique is the employment of weighted Hamming distance in calculating the distance between two states. In estimating how far the current circuit state is from the target, we constantly need to compute the distance of two states. Previously, Hamming distance has been used, which counts the number of bits differed in the two given states. However, reported results on this metric suggested that the Hamming distance does not necessarily correlate with the actual distance in state space [16]. Therefore, we enhance it by weighting the bits in the state by their controllability. Intuitively, the difference in a hard-to-control FF will contribute more than a easy-to-control FF to the overall distance of the two states. There are two cases to be differed here when calculating the difference:

- the bit in candidate state is 0 and in target state is 1;
- the bit in candidate state is 1 and in target state is 0.

In the first case, we need a 0 to 1 transition in the FF bit to justify the FF value and a 1 to 0 transition is needed in the second case. Therefore, the distance is target-state-related. We define the state distance weight according to the target state’s value:

$$Definition 3. Distance weight for i^{th} FF in the state is DSW_i = (1 - FFCC_i) \times n_{FF}.$$ 

where $FFCC_i$ is the controllability of $FF_i$. If the target state value in this bit is 0, then it is the 0 controllability of this FF. If target state value in this bit is 1, then it is the 1 controllability. $n_{FF}$ is the total number of FFs. The 0 and 1 controllability used here are the probabilistic based measurement instead of SCOAP so that their value are always between 0.0 and 1.0. In practice, we use the frequency of each FF switching to 0 and 1 during simulation as its 0 and 1 controllability. Therefore, the easier the FF is to control 0 or 1, the close its corresponding switching frequency value is to 1.0, and from definition 3, the small the weight for this FF. $n_{FF}$ serves here as the upper bound for the weight. The worse case is that a FF is never switched but 0/1, then its corresponding $FFCC_i$ is 0, then its weight is the max weight $- n_{FF}$. Based on the weight defined for each FF, the overall weighted Hamming distance between state A and target state B is computed using the following formula:

$$WHD(A, B) = \sum_{i=1}^{n_{FF}} |FF^A_i - FF^B_i| \times DSW_i,$$

where $FF^A_i$ and $FF^B_i$ are the bit value for the two states respectively. WHD denotes “Weighted Hamming Distance”. From equation (2) we can see that the original Hamming distance is the case in the weighted Hamming distance that all $DSW_i = 1$.

4. OVERALL PROCEDURE

We describe the overall procedure of the simulation-based ATPG for safety property verification in this section. We assume that each safety property has been transformed to a “target fault” using the monitor circuit introduced in Section 2. The procedure is given below.
1. Compute reset power, calculate reset mask probability
2. Build initial state table by simulating 1000 random vecs
3. State table enrichment phase: enrich the state table by generating & simulating another 1000 vecs, using GA & reset signal masking to maximize # of state visited.
4. For each target fault mapped from the target property
   4.1 Call PODEM to generate a combinational test;
   4.2 Justify the state part of the combinational test $S_{tar}$:
      nFail = 0; nLocalFail = 0;
      while (nFail <= nLimit), start from the current state,
      4.2.1 Initialize GA population with random seeds;
      Seed the GA any past moves that took the ckt to a state $S_p$, such that $WHD(S_{tar}, S_p) \leq d$;
      4.2.2 Use GA w/ threshold value simulation to evolve this population and picks the best candidate;
      4.2.3 Make the next move: simulate the best candidate, update the current state;
      4.2.4 If (current state == target state), success: goto next property.
      else
      fail: nFail++;
      if fail at the same local minima
      nLocalFail++;
      end if
      if (nLocalFail == nLocalLimit)
      generate a random seq as next move,
      nLocalFail = 0, goto 4.2.1
      end if
      end if
      end while

The main purpose of introducing steps 2 and 3 is to enrich the state table. The test generator maintains a state table which stores every visited state and the vector sequence (“move”) that took the circuit to this state. All moves are concatenated to form the final test set. Every time after GA optimization, the test generator will append the best candidate in GA as the “next move” to the test set. The enrichment of state table will help because step 4.2.1 depends on the richness of this table. In step 4.2.1, some past moves are selectively seeded into GA. Their role is as an educated guess of initial values for the optimization algorithm. The more information we have in the state table, the more seeds we would have and hence the greater the chance that the GA will succeed. This state table and the associated moves are the important knowledge learned on-the-fly by the test generator.

4.1 Local Minima

As noticed from the procedure, our state space search treats the state justification problem as a large-scale optimization problem. Since the objective generally cannot be reached in a few steps, heuristics (threshold value fitness, weighted Hamming distance) are used to guide the search to approach the target step by step. GA is in charge of local optimization to decide the next move. As in all greedy algorithms for solving large-scale optimization problems, our procedure can be stuck into local minima as well, which is manifested by reaching a state closest to the target state and has a fitness higher than all its nearby neighbors.

To solve the local minima problem, we introduce in step 4.2.4 a counter $nLocalFail$. If the search has been trapped into the same local minima for a number of times $nLocalLimit$, then a random sequence is accepted as the next move, although its fitness might be much lower than the best one in GA. With this random walk, we hope the search can escape from the local minima. This technique is also known as hill climbing in optimization methods.

5. EXPERIMENTS AND RESULTS

<table>
<thead>
<tr>
<th>circuit name</th>
<th>total # of gates</th>
<th>total # of FFs</th>
<th>total # of properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>s5378</td>
<td>3043</td>
<td>179</td>
<td>9</td>
</tr>
<tr>
<td>s38417</td>
<td>26274</td>
<td>1636</td>
<td>50</td>
</tr>
<tr>
<td>s38584</td>
<td>23008</td>
<td>1452</td>
<td>50</td>
</tr>
<tr>
<td>mma</td>
<td>7166</td>
<td>327</td>
<td>1</td>
</tr>
</tbody>
</table>

We implemented the aforementioned techniques in a total of 21,000 lines of C++. We called this new tool “Threshold value simulation guided VERifier” (THRIVE). THRIVE was tested on some large ISCAS89 benchmark circuits s5378, s38417, s38584 and an industrial circuit mma. All these circuits have large numbers of FFs which can potentially make the formal verification methods that involve BDDs difficult to handle. The safety properties to be verified for ISCAS89 circuits are random conjunctions of internal signals of the circuits. The one property for mma is, however, a real property from the designer, which is known to be true.

We compared our results with four tools: HITEC [17], BINGO [18], CHAFF [6] and the original STRATEGATE [9, 10]. HITEC is the most widely used deterministic ATPG tool in academia. BINGO is an efficient symbolic model checking tool for verifying realistic properties of sequential circuits and protocols. CHAFF is a SAT solver. Due to a careful engineering of all aspects of search, CHAFF has achieved significant performance gains over conventional SAT solver and is considered as the state-of-the-art. To apply CHAFF to verification, each property to be verified is converted to a CNF formula by unrolling the circuit $k$ time frames. We did the unrolling with an incremental step of 20 time frames each time and up to 100 time frames in total: if the property cannot be verified within $k$ time frames, then $k+20$ is tried. STRATEGATE is currently the state-of-the-art simulation-based ATPG tool, which has achieved significant performance gain over HITEC in test generation. The HITEC and BINGO experiments were run on a Ultra-SparcII, 200MHz with 256M RAM. All other experiments were run on a Pentium III 1Ghz clock, 1024M bytes RAM machine installed with Redhat Linux 7.1.

Table 2 reports the characteristics of the circuits, including the total number of gates, total number of FFs and total number of properties to be verified for each large circuit experimented. Verification results are shown in table 3, in which we compared the results for HITEC, BINGO, CHAFF, STRATEGATE with THRIVE, proposed in this paper. For each circuit, the number of properties is first listed, followed by the number of verified properties and the execution time taken to verify them. As shown in the table, THRIVE achieved impressive performance over other techniques. For example, in s5378, HITEC aborted all nine properties. BINGO can verify all the properties, however, with large computational costs. CHAFF verified all nine of them in a very short time (10.3 sec). STRATEGATE can verify only 1 of the 9 properties to be false while THRIVE can verify all 9 properties to be false, though taking longer.
### Table 3: Verification Results

<table>
<thead>
<tr>
<th>ckt</th>
<th>prop.</th>
<th>HITEC verified time(s)</th>
<th>BINGO verified time(s)</th>
<th>CHAFF verified time(s)</th>
<th>STRATEGATE verified time(s)</th>
<th>THRIVE verified time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s5378</td>
<td>9</td>
<td>0 aborted</td>
<td>9 36835</td>
<td>10.3 1</td>
<td>58  1</td>
<td>2063</td>
</tr>
<tr>
<td>s38417</td>
<td>50</td>
<td>2 2913</td>
<td>fail to build BDD</td>
<td>12 8805</td>
<td>9  4178</td>
<td>18 8957</td>
</tr>
<tr>
<td>s38584</td>
<td>50</td>
<td>2+10 4563</td>
<td>fail to build BDD</td>
<td>31 11797</td>
<td>27 3852</td>
<td>27+2 5771</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mma</td>
<td>1</td>
<td>0 aborted</td>
<td>1 7442</td>
<td>1 374</td>
<td>0 aborted</td>
<td>1 189</td>
</tr>
</tbody>
</table>

6. CONCLUSION

We have presented an effective technique on safety property checking using simulation-based ATPG. We proposed several heuristics, namely, small deterministic ATPG, reset signal masking, threshold value simulation and weighted Hamming distance, to effectively guide the simulation engine in the large state space to reach the target state. These enhancements help the simulation engine to avoid revisiting previously visited states, deterministically obtain a combinational solution, and refine the cost function during search. Experiments on large benchmark and real industrial circuits showed that our simulation-based verifier can verify more properties which previous techniques failed in comparable execution time.

7. REFERENCES


