Efficient Reachability Checking using Sequential SAT

{gpartha,madiyer,timcheng,licwang}@ece.ucsb.edu
Department of ECE, Room 4108,
University of California – Santa Barbara
Santa Barbara, CA 93106

Abstract – Reachability checking and Pre-image computation are fundamental problems in ATPG and formal verification. Traditional sequential search techniques based on ATPG/SAT, or on OBDDs have diverging strengths and weaknesses. In this paper, we describe how structural analysis and conflict-based learning are combined in order to improve the efficiency of sequential search. We use conflict-based learning and illegal state learning across time-frames. We also address issues in efficiently bounding the search space in a single time-frame and across time-frames. We analyze each of these techniques experimentally and demonstrate the advantages of each technique. We compare performance against a commercial sequential ATPG engine and VIS [13] on a set of standard benchmarks.

I. INTRODUCTION

Given a set of Boolean value assignments in a sequential circuit, and an initial state, the Sequential SAT problem is to find a sequence of input vectors to the circuit, such that the value assignments are satisfied or to prove that no such sequence exists. Sequential ATPG, image-computation, and reachability checking can be reduced to sequential SAT. Marchok et al. [10], showed that backward reachability checking takes up-to 90% of the time in sequential ATPG. Image/Pre-image computation is the main run-time bottleneck in formal verification [3]. Therefore, sequential SAT is a fundamental problem in ATPG and formal verification.

ATPG has traditionally relied on structural methods to check if a state can be reached from an initial state (i.e., it is legal), but it can take from excessive time on some problems. Image computation for formal verification has traditionally been done by BDD-based methods, which can take excessive memory for some problems. ATPG typically use structural techniques for a depth-first search (DFS) in the state space for reachability checking. Image and Pre-image computation corresponds to an implicit breadth-first search (BFS) in the state space, forward and backward respectively.

There have been many efforts aimed at exploiting circuit structure information for pure BDD-based image computation [11]. There have also been some efforts on combining SAT and BDDs with limited success [5] [6]. However, to date, there has been very little work on leveraging structural search and advanced SAT techniques for reachability checking and pre-image computation. A SAT-based technique is memory-efficient since the transition relation is never explicitly constructed as compared to OBDD-based methods [11]. Once the OBDD(s) of a transition relation is constructed, pre-images can typically be constructed quickly by existential quantification, if the memory explosion problem is not hit. A generalized SAT-based technique would take more time since it has to perform search to enumerate all states in a particular pre-image. Essentially, SAT and OBDDs can have orthogonal time-space complexity characteristics.

In this paper, we describe several techniques for improving the time-behavior of sequential SAT – both for reachability checking and pre-image computation. The main contributions of the research in this paper are as follows:

1. True sequential search on a single time-frame at a time, using implicit time-frame expansion.
2. Conflict-based sequential learning.
3. Efficient state cover checking using a SAT database
4. Efficient Sequential Bounding with minimal state cubes and a unique state-space avoiding clause scheme.
5. Analysis of sequential SAT for reachability checking and pre-image computation.

Paper Outline The rest of this paper is organized as follows: We give a brief overview of related prior-art in Section II. We describe some of the background relevant to this paper in Section III. In Section IV we describe the mechanics of general sequential search. We present the basic algorithms for sequential SAT in Section V. We describe techniques for reducing the size of intermediate solutions in sequential SAT in Section VI. In Section VII, we describe techniques for using these intermediate solutions to bound the search space efficiently. In Section VIII, we present experimental results on some standard benchmarks to demonstrate the speed and capacity performance of sequential SAT as compared to a state-of-the-art commercial ATPG engine, and a model-checker – VIS(v2.0) [13]. Finally, we present our conclusions in Section IX.

II. RELATED WORK

There exists a large body of work on techniques, which effectively use the structure of the circuit to perform branch-and-bound search. Cheng et al., [7] presents a good survey of the current work in traditional structural sequential search. SAT based techniques and implication graph based techniques have been used to speed up the basic combinational part of the problem [2]. A variety of techniques have been proposed for illegal state identification and storage to prune the state space effectively [6].

Gupta et al., [5] proposed a method of combining OBDDs and SAT using separator-set partitioning to allow SAT to decompose the search space and OBDDs to compute all solutions below intermediate points in the SAT decision tree. McMillan [11] proposed a method for quantification of variables in a CNF formula for image computation using search. Kang and Park [8] proposed a similar method using two-level minimization for reducing the set of blocking clauses. Both these methods rely on Boolean SAT for exploring the search space. However, our method uses a 3-valued algebra which
significantly increases the number of states visited in a single search step. Iyer et al., [12] described a general sequential SAT engine that uses structural analysis and conflict-based learning. We extend that work in this paper, with algorithmic improvements to the state learning, new search strategies, and analysis of reachability checking, and pre-image computation using sequential SAT.

The main components of the run-time complexity of pre-image computation using sequential SAT are the number of state cubes found at each step and the combinational search required to find each of these cubes. Sheng et al., [13], describes a method for one-step pre-image computation based on PODEM [7] and a variant of EST [6] based on equivalent cut-sets, in order to reduce the time spent in enumerating states for a pre-image. The pre-image is stored as a free BDD. They showed results for a single pre-image computation as compared to an OBDD based approach. However, their approach is limited by the fact that the size of their pre-images is large, which can take long run-times for n-step pre-image computation.

III. PRELIMINARIES

In this section, we briefly describe the key concepts of generic SAT algorithms and state space search used in this paper. Interested readers may refer to e.g. [9,17] for a more detailed description.

Given a finite set of variables, \( \mathcal{V} \), over the set of Boolean values \( \mathcal{B} \in \{0, 1\} \), a literal, \( l \in \mathcal{L} \) is a variable, \( v \lor \neg v \in \mathcal{V} \). A clause \( c_i \) is a disjunction of literals. A formula \( f \), is a conjunction of clauses. An assignment, is a mapping, \( \mathcal{V} \Rightarrow \mathcal{B} \). Following the convention in [9,17], we equate an assignment \( A \) with a clause. For example, the assignment \( \{v_1(0), v_2(1)\} \equiv (v_1 + \overline{v_2}) \).

A conflict is the simultaneous implication of opposite Boolean values on a variable. Given a conflict at a variable \( v_i \), a conflicting assignment is a subset \( a \subseteq A \) of an assignment \( A \), such that the \( f(a) \) is sufficient to cause the same conflict at \( v_i \). A conflict clause is the clause corresponding to \( a \).

A finite state machine (FSM) \( M \) is defined as a 6-tuple \( M = (Q, \Sigma, O, \delta, \lambda, I) \), where \( Q \) is a finite set of states, \( \Sigma \) is the input alphabet, \( O \) is the output alphabet, \( \delta : Q \times \Sigma \rightarrow O \) is the output function, and \( I \) is the set of initial states. A synchronous sequential circuit can be represented as an FSM. The state transition function \( \delta : Q \times \Sigma \rightarrow O \) determines the next state of the FSM or circuit depending on the the current state and inputs. If \( x \equiv \{x_1, \ldots, x_m\} \) are the primary inputs (PIs), \( s \equiv \{s_1, \ldots, s_n\} \) are the state variables (PPIs), \( \delta \equiv \{\delta_1, \ldots, \delta_n\} \), and \( s^+ \equiv \{s^+_1, \ldots, s^+_n\} \) are 1-step reachable states, then we can define the transition relation \( T \) as follows:

\[
T(s^+, s, x) = \bigwedge_{k=1}^{n} (s^+_k \oplus \delta_k(s, x)), \text{ where } \oplus = \text{XNOR}
\]

A state cube is a tuple \( (v_{i_1}, \ldots, v_{i_k}) \), where \( \{i_1, \ldots, i_k\} \in \{1, \ldots, n\} \) and \( k < |\text{state variables}| \). A state cube is called a legal state cube if the corresponding states covered by the cube are legal; it is called an illegal state cube if the covered states are illegal. The size of a state cube is the number of states that are covered by the cube. A state clause is a clause formed from a set of Boolean value assignments on state variables following the convention above. A state frontier \( F_i \) is defined as the states that can reach the objective state in exactly \( i \) transitions.

A pre-image computed using sequential SAT can be defined as

\[
\text{PreImage}(T, F) = \{s | \exists t \in F, such that T(s, t) = 1\}
\]

where \( F \) is the state frontier, which represents all the states visited so far. If \( R_i \) is the set of states that have been visited by BFS search for \( i \) time-frames, then iterative pre-image computation can be represented as follows:

\[
R_{i+1} = R_i \cup F_i
\]

\[
F_{i+1} = \text{PreImage}(T, F_i) - R_{i+1}
\]

In the next section, we describe search in sequential circuits.

IV. SEQUENTIAL SEARCH

In this section, we discuss the two major search strategies in sequential SAT. The figure 1 shows backward search in the sequential space of an example circuit, where the nodes correspond to state cubes and the edges represent existence of an input vector \( i \), that can take the circuit from one state cube \( (s_i) \) to another \( (s_{i+1}) \). Figure 1 shows all the cubes that can be found when searching for a solution to \( \text{Obj} \), which is a set of Boolean assignments on internal circuit lines. A DFS approach would search the state space path-by-path, while a BFS approach would enumerate all the states in a particular state frontier, before moving to the next one.

DFS and BFS strategies have some orthogonal advantages and disadvantages.

1. Overhead: DFS incurs overhead with no appreciable gains, when dynamic illegal state identification [12] is performed, since the intermediate search results per time-frame have to be stored. However, for reachability analysis, we do not store the intermediate results for DFS, but restart search for every new state cube.

2. Bounding techniques: Unreachable states can be used to bound the search efficiently. BFS requires that we finish search for each state before we can determine whether it is
unreachable, while it is possible to do this earlier in DFS. Hence, we can potentially bound the search space faster than in BFS, by the techniques in \([12]\). In practice, this is offset by the overhead inherent to those techniques. BFS search can be improved by minimizing intermediate state frontiers, as explained in Section[V].

3. Faster termination: DFS typically terminates faster than BFS, if (a) the problem is satisfiable and (b) the legal state space is large. However, if the legal state space is small or if the problem is unsatisfiable, BFS performs much better since it has lower overhead and both algorithms have to explore comparable search spaces.

4. Shorter state sequences: BFS finds the shortest state sequence that can satisfy the objective. DFS can find much longer state sequences depending on the heuristics used to guide the search.

In the next section, we describe the basic algorithms in sequential SAT.

V. SEQUENTIAL SAT

Sequential SAT explores the state space using the search strategies described in the previous section. Sequential SAT is memory efficient since it operates on a single copy of the next-state logic of a given state machine as shown in Figure[2] which corresponds to a time-step in the Figure[1]. The next state logic that updates the registers represents the transition function \(\lambda\) defined in Section[III].

![Fig. 2. Single Time-frame of a Sequential Circuit](image)

The two fundamental operators that we need in order to do sequential search are a) Finding a set of states (cube) that can reach the current set of states in one transition (1-step backward reachable), and b) Make sure that visited state cubes are not re-visited.

New states are “discovered” using combinational SAT. Given an objective \(\text{Obj}\), we generate an assignment on the previous state lines and primary inputs that can satisfy \(\text{Obj}\), using 3-valued search \([12]\). This procedure will generate a set of backward 1-step reachable states if \(\text{Obj}\) is a state cube. We then iteratively search time-frame by time-frame to check whether the current satisfying state cube is legal. The iteration loop determines whether the search will be DFS or BFS in the state space.

The general factors in the efficiency of sequential SAT are the following:

1. State cube reductions: Sequential search is more efficient when the cubes found are larger, i.e., the more states each cube covers, since there are a smaller number of cubes to be classified.

2. State space bounding: This is done by the use of state back-tracking clauses, which avoid revisiting any of the states covered by the clause. New solution cubes can cover other cubes in the same time-frame. For example, \(c^2\) is covered by \(c^3\) in Figure[1].

It is more efficient to use BFS search for pre-image computation, since we can minimize the state frontiers as described in the next section. DFS or BFS search can be used for reachability checking. In the following sections, we describe several techniques for addressing each of the efficiency issues above.

VI. STATE CUBE REDUCTIONS

We summarize an algorithm which was proposed in \([12]\) to find a minimal state-clause for justification. We describe an efficient method for checking covers in the state cache. We also describe the advantages of using two-level minimization \([14]\) technique for reducing the number of cubes found during search.

A. Maximizing State Cubes

It would be best to find the smallest assignment that satisfies an objective, since the corresponding state cube is the largest, which in turn affects efficiency. This is applicable for both DFS and BFS.

We now briefly describe REDUCE ASSIGNMENTS, which takes as inputs a topologically sorted circuit \(Ckt\), an assignment vector \(A\), and a set of objectives \(obj\); and produces a smaller assignment \(a\), which satisfies \(obj\). The algorithm picks sets of PIs and PPIs depending on whether they are controlling or non-controlling values with a cost function that minimizes the total number of PPIs picked. The reader is referred to \([12]\) for further details. The algorithm is illustrated in the Figure[2] for clarity.

![Fig. 3. Example of REDUCE ASSIGNMENTS.](image)

Assume that the decision procedure sets the values of \(\{G0, G1, G2, G3, G5, G6, G7\} = \{1011001\}\), which satisfies the objective of \(G14 = 1\). The gates in the input cone of \(G14\) are marked in bold. The traversal starts at the PIs and PPIs with unit lists, which are propagated forward. When the traversal reaches \(G8\), the algorithm chooses \(G0\), since it is a PI over \(G6\), which is a PPI, though both are at the same value. When the traversal reaches \(G15\), the input lists are merged since \(G15\) is at a non-controlling value of 0. There are no more merges till we reach the objective site and the final set consists of \(G0, G7\).
It has been shown that minimizing assignments by this heuristic can yield improvements of $3x$–$300x$ in run-time and up to $5x$ in length of satisfying sequences [12].

**B. Minimizing the Number of State Cubes**

The number of state cubes that have to be checked for reachability at an intermediate step is minimized in order to reduce the combinational search that is done by compressing the set of state cubes in a pre-image before checking for the next pre-image.

Our formulation implicitly avoids repeating state cubes by sequential backtracking clauses and state caching. Each of these techniques have different levels of importance in DFS and BFS.

**B.1. Minimizing State Frontiers**

BFS search efficiency depends on the size of the intermediate state frontiers as defined in Section 2. All solutions for every cube $s_j, j \in \{0, \ldots, |F_i|\}$, in a given frontier $F_i$, have to be computed for the next pre-image. Every extra cube in $F_i$ requires a non-trivial combinational satisfiability check.

The solutions for a cube $s_j$ can cover or be covered by the solutions for another cube $s_k, k \neq j$. Also, the cubes themselves may be collapsible, since cubes of the form $(a+b+c)$ and $(a+b+c)$ may exist in $F_i$. While the former is mitigated by the lookup with covering in the state cache, the overhead of the lookup still exists.

We use two-level minimization techniques [14] to reduce the min-terms of the function represented by the cubes in $F_i$. The reduced set of min-terms are guaranteed to cover exactly the same states as the original set of min-terms. These are then used as the starting set for calculating the next pre-image.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Img</th>
<th>State Cubes</th>
<th>Run-Time(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abp (1)</td>
<td>8</td>
<td>1849</td>
<td>149</td>
</tr>
<tr>
<td>Bakery (3)</td>
<td>15</td>
<td>4408</td>
<td>303</td>
</tr>
<tr>
<td>Bakery (4)</td>
<td>15</td>
<td>2449</td>
<td>189</td>
</tr>
<tr>
<td>Bakery (5)</td>
<td>60</td>
<td>49495</td>
<td>1072</td>
</tr>
<tr>
<td>Crc (1)</td>
<td>6</td>
<td>&gt;200K</td>
<td>16</td>
</tr>
<tr>
<td>Cups (1)</td>
<td>11</td>
<td>1311</td>
<td>333</td>
</tr>
<tr>
<td>DAIO (2)</td>
<td>66</td>
<td>15618</td>
<td>13473</td>
</tr>
<tr>
<td>DAIO (3)</td>
<td>33</td>
<td>7190</td>
<td>6122</td>
</tr>
</tbody>
</table>

**Analysis**

The experiments shown in Table 1 show the results of cube-reduction for pre-image computation. The benchmarks are from the VIS benchmark set and were synthesized to a gate-level net-list using VIS [13]. Each benchmark was checked for safety properties using BFS sequential search. Correctness of results was verified by cross-checking with VIS and simulating the witness vectors for False properties. Column 2 shows the number of pre-images that were calculated before classifying the property as True or False. Columns 3 and 4 show the total number of cubes with and without minimization. Columns 5 and 6 show the overall run-time with and without cube minimization.

As we can see from Columns 5 and 6 in Table 1 the total number of cubes for the pre-images was reduced by a factor of 1.2x to 1000x+ by two-level minimization. This led to a corresponding decrease in run-time by approximately the same factor. Note that there seems to be a direct correlation between the number of cubes and the run-time since the decrease in the number of cubes is directly proportional to the decrease in run-time. It is clear from the experiments that the number of cubes per image is a key factor in run-time efficiency, and that cube minimization is a low-cost method for speeding up sequential SAT.

**C. Covering in the State Cache**

Efficient cover checking in the state cache is desirable for any algorithm that requires multiple look-ups in the state cache, like that in state frontier minimization as described in the previous section.

Traditional approaches that do loop checking by caching visited states, flag a newly detected state as revisited if it corresponds exactly with a state already present in the state cache. However, we know that states that can be reached by a cover state already visited will be a superset of the states that can be reached by the current covered state. It is sufficient to check for reachability of the cover state rather than the current covered state. Unfortunately, the worst-case complexity of checking for covers by traditional methods is $O(n^2)$, since each cube has to check against the new cube. We describe a novel implication-based method for checking for covers in the state clause database that is efficient and simple to implement.

The state cache stores the visited states as a clause database. Given an assignment $A_i$ that satisfies the objectives for time-frame $t_i$, the recorded clause is:

$$s_i \equiv \text{ReduceAssignments}(A_i) \setminus \{v_i \in \text{PIs}\}$$

A newly discovered state clause $S_{c}$, is covered by a clause $S_{i}$ in the state cache, if $S_{c} \subseteq S_{i}$. For example, if $S_{c} = (a+b+c)$, and if $S_{i} = (a+b)$ exists in the state cache, then $S_{i}$ is returned as a hit in the state cache. Note that the state clauses are defined according to the convention in Section 2.

The implication-based lookup works as follows. On finding the new state cube $S_c = (a+b+c)$ corresponding to the assignment $\{a, b, c\} = \{1, 1, 0\}$, we find the implications of the values $\{a = 1, b = 1, c = 0\}$ in the state cache. The first clause that has a conflict is returned as a covering clause. In this case, it will be $S_i$, since it implies a value $b = 0$, when $a = 1$, which conflicts with the assignment $b = 1$. The procedure is guaranteed to return the largest cover (smallest clauses), since smaller clauses will imply values before larger clauses.

The time complexity of this procedure is $O(v.k)$, where $v$ is the number of variables assigned in the new state and $k$ is the number of clauses in the state cache that have at least one of the variables in the new state cube.

In the next section, we describe how we efficiently bound the search space for sequential SAT.

**VII. STATE SPACE BOUNDING**

Our fundamental approach to state space bounding is through sequential backtracking. This is done by creating a state back-tracking clause $b$, corresponding to the literals in current state cube. This creates a constraint that the last satisfying assignment cannot be repeated [12]. With the addition of back-tracking clauses, all decisions that lead to the same
state-cube that are already explored in the same time-frame are avoided or detected before a new state-cube is found by implication of these clauses. Earlier approaches [6] relied on using an expensive, lookup-table based scheme to avoid revisiting decision spaces. The clause-based scheme saves the cost of lookup and is correspondingly more efficient. We describe how we use the state back-tracking clauses in DFS and BFS below:

A. DFS Search Bounding

In our earlier work [12], sequential SAT using DFS search checked for loops by checking a new state assignment against the state cache, which creates a backtrack if a hit is returned. Several optimizations for illegal state identification were described. However, for the purpose of reachability checking, it is not necessary to visit every edge in the state graph. It is sufficient to visit every state. Hence, we use state back-tracking clauses to ensure that states not revisited. This saves us the overhead inherent in doing early illegal state classification.

B. BFS Search Bounding

BFS in the state-space corresponds to a Pre-image computation [11]. In sequential SAT, the set of reached states at time-step $i$, $R_{i+1}$ (Eqn 2) is the set of state cubes that have been visited so far. So, the state frontier, $F_{i+1}$ (Eqn 3) reduces to searching the space represented by the next-state logic under the constraints that no state in $R_{i+1}$ should be revisited. This is simply done by adding all state backtracking clauses to avoid generating any solutions that correspond to these states. So, we can modify the transition function of the sequential circuit as:

$$\lambda_{i+1} \equiv \lambda_i \land \bigwedge_{j=0}^{m} s_j = |F_i|, s_j \in F_i$$

This formulation implicitly avoids all value assignments that lead to the states covered by the state-cubes. All solution cubes for the current time-frame are part of the current pre-image. The current pre-image is completed, when no more solutions exist for the current time-frame. This formulation results in sets of cubes that implicitly enumerate each pre-image.

C. Reuse of Back-tracking Clauses

The state back-tracking clauses corresponding to unreachable states can be re-used in further reachability analysis on the same circuit. However, they can be safely used only in UNSAT cases, where the original state objectives are unreachable from the initial state. A state back-tracking clause implicitly create the constraint that the states covered by the clause should not be revisited, during search. This means that edges from un-visited states to the states covered by the back-tracking clause are never explored. If state clauses from a SAT case is re-used, the results of further SAT checks may be incorrect, since the clauses may overly restrict the search space. Therefore, we do not classify any of the states in a SAT case as truly illegal, and only re-use state clauses that are known to be truly unreachable from the initial state i.e., in an UNSAT case.

In the following section, we describe experimental analysis of sequential SAT for reachability checking and pre-image computation.

VIII. EXPERIMENTS

The experiments shown in this section analyze the following:
1. Sequential SAT for reachability checking
2. Sequential SAT for pre-image computation.

All the experiments were run on a 2.0 GHz Intel Pentium-4 machine with 1 Gigabyte of RD-RAM running Linux and a Sun UltraSparc-3d machine running Solaris 8.0. The results were scaled using an independent set of experiments so that the run-times can be directly compared. It should be noted that the commercial ATPG engine was run in justification mode, which makes it equivalent to sequential SAT. VIS(v2.0) was used with the standard scripts in the distribution.

A. Reachability Checking versus ATPG

The first set of experiments conducted are targeted at checking the efficiency of backward reachability checking using sequential SAT versus a commercial ATPG engine. The limits set were a million back-tracks, 300 cycles and 3600 secs per property. The objectives were a Boolean value of 0 and 1 on each line in a set of examples from the ISCAS’89 benchmarks. This is intended to test both engines intensively, since most of these objectives can be checked relatively easily. However, with enough instances, we should be able to make some reasonable comparisons. The results are shown in Table II. Column 2 shows the total number of objectives classified as SAT/UNSAT/Aborted for sequential SAT and the commercial engine respectively. Columns 4 and 6 show the run-times for each method.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Total</th>
<th>Sequential SAT</th>
<th>Comm. ATPG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Obj</td>
<td>Result&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Time</td>
</tr>
<tr>
<td>s444</td>
<td>362</td>
<td>358/40</td>
<td>0.95</td>
</tr>
<tr>
<td>s510</td>
<td>422</td>
<td>61/361/0</td>
<td>2.54</td>
</tr>
<tr>
<td>s526</td>
<td>386</td>
<td>381/5/0</td>
<td>1.48</td>
</tr>
<tr>
<td>s953</td>
<td>790</td>
<td>331/459/0</td>
<td>0.74</td>
</tr>
<tr>
<td>s5378</td>
<td>5558</td>
<td>5015/5358/26286</td>
<td>4953/282/323</td>
</tr>
<tr>
<td>s35932</td>
<td>32130</td>
<td>32130/0/0</td>
<td>704.25</td>
</tr>
</tbody>
</table>

<sup>1</sup> Completed Results: num SAT/num UNSAT/num Aborted

The results show that sequential SAT demonstrates upto 4 orders of magnitude speedup on some of these testcases. Sequential SAT also is more robust than the commercial tool since it finishes classification correctly on almost all the test-cases, except for s5378 where it aborts on 8 objectives. It appears that sequential SAT is considerably more efficient and robust than current structural sequential SAT techniques which do not use extensive learning.

B. Reachability Checking versus BDDs

BDD-based approaches are extremely efficient for image and pre-image computation, provided they do not run out of memory. In the next set of experiments, we compare sequential SAT in both DFS and BFS search modes against VIS on several benchmark circuits from the VIS tool distribution. We compare the efficiency these methods on reachability checking and pre-image computation for safety prop-
properties. VIS was run in two modes: (1) forward and backward reachability (default), and (2) backward reachability only.

The gate-level net-lists were synthesized using VIS. The results are shown in Table III. Column 1 shows the benchmark name. Column 2 shows the number of False and True properties in each benchmark. Columns 3 and 4 show the run-time for sequential SAT in DFS mode and the number of aborts. Columns 5 and 6 show the run-time for sequential SAT in BFS mode and the number of aborts. Columns 7-10 show the run-time and number of aborts for VIS in the two modes mentioned above. Both tools were given a time-out limit of 15000 seconds and no other limits.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#props</th>
<th>SSAT</th>
<th>Abt</th>
<th>SSAT</th>
<th>Abt</th>
<th>VIS</th>
<th>Abt</th>
<th>VIS</th>
<th>Abt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Am2901</td>
<td>1/1</td>
<td>10.63</td>
<td>0</td>
<td>-to-</td>
<td>1</td>
<td>-to-</td>
<td>1</td>
<td>768.66</td>
<td>0</td>
</tr>
<tr>
<td>Am2910</td>
<td>0/4</td>
<td>5.68</td>
<td>0</td>
<td>0.17</td>
<td>0</td>
<td>0.17</td>
<td>0</td>
<td>5.68</td>
<td>0</td>
</tr>
<tr>
<td>Arbiter</td>
<td>0/1</td>
<td>9.1</td>
<td>0</td>
<td>0.04</td>
<td>0</td>
<td>0.04</td>
<td>0</td>
<td>9.1</td>
<td>0</td>
</tr>
<tr>
<td>Bakery</td>
<td>0/0</td>
<td>32.47</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>32.47</td>
<td>0</td>
</tr>
<tr>
<td>Barrel</td>
<td>0/0</td>
<td>32.47</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>32.47</td>
<td>0</td>
</tr>
<tr>
<td>Coherence</td>
<td>0/0</td>
<td>14.46</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>14.46</td>
<td>0</td>
</tr>
<tr>
<td>Chamele</td>
<td>0/0</td>
<td>14.46</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>0.09</td>
<td>0</td>
<td>14.46</td>
<td>0</td>
</tr>
<tr>
<td>CRC</td>
<td>1/0</td>
<td>768.66</td>
<td>0</td>
<td>768.66</td>
<td>0</td>
<td>768.66</td>
<td>0</td>
<td>768.66</td>
<td>0</td>
</tr>
<tr>
<td>Cups</td>
<td>2/3</td>
<td>17.55</td>
<td>0</td>
<td>0.17</td>
<td>0</td>
<td>0.17</td>
<td>0</td>
<td>17.55</td>
<td>0</td>
</tr>
<tr>
<td>Diao</td>
<td>3/1</td>
<td>60.95</td>
<td>0</td>
<td>54.73</td>
<td>0</td>
<td>54.73</td>
<td>0</td>
<td>60.95</td>
<td>0</td>
</tr>
<tr>
<td>Iftpch</td>
<td>6/6</td>
<td>12.47</td>
<td>0</td>
<td>12.47</td>
<td>0</td>
<td>12.47</td>
<td>0</td>
<td>12.47</td>
<td>0</td>
</tr>
<tr>
<td>b13</td>
<td>23/20</td>
<td>12317.5</td>
<td>0</td>
<td>12317.5</td>
<td>0</td>
<td>12317.5</td>
<td>0</td>
<td>12317.5</td>
<td>0</td>
</tr>
</tbody>
</table>

VIS \(_{FB}\) with forward reachability check enabled
VIS \(_{B}\) with only pre-image computation enabled
* x/y are \#False/\#True properties

As we can see from the table, sequential SAT is superior to VIS in some cases. Sequential SAT in DFS mode can complete on cases where VIS takes a large amount of time, for example, Am2901, Am2910 and CRC. In these cases, VIS aborts when forward reachability is enabled. However, it completes on them when only pre-image computation is done. But, this strategy fails on Coherence, which can be completed only with forward reachability enabled. It is also evident that VIS is considerably faster in pre-image computation in most cases.

It is interesting to note that the behavior of sequential SAT in BFS mode closely mimics that of VIS doing only pre-image computation. It is clear that, in general, a search strategy for sequential SAT in BFS mode closely mimics that of VIS doing only pre-image computation. We have demonstrated that the search strategies in sequential SAT can be enhanced considerably to complement or even compete with OBDD-based model checking. In our future work, we will explore different search strategies in sequential SAT for improving efficiency.

**References**


