SATORI – A FAST SEQUENTIAL SAT ENGINE FOR CIRCUITS

M. K. Iyer G Parthasarathy K.-T. Cheng

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

ABSTRACT

We describe the design and implementation of SATORI – a fast sequential justification engine based on state-of-the-art SAT and ATPG techniques. We present several novel techniques that propel SATORI to a demonstrable 10x improvement over a commercial engine. Traditional sequential justification based on ATPG or, on a bounded model of the sequential circuit using SAT, has diverging strengths and weaknesses. In this paper, we contrast these techniques and describe how their strengths are combined in SATORI. We use conflict-based learning in each time-frame and illegal state learning across time-frames. This enables both combinational and sequential back-jumping. We experimentally analyze the main features of SATORI by comparing SATORI's performance against a state-of-the-art SAT solver – ZCHAFF [13] using a bounded model, and a commercial sequential ATPG engine performing justification. Additional results are presented for SATORI versus the commercial ATPG engine and VIS [16] on ISCAS '89 and ITC'99 benchmark circuits for an application to assertion checking.

1 Introduction

Sequential SAT or *justification* is the problem of finding an *ordered sequence* of input assignments to a sequential circuit, such that the desired objective is satisfied, or proving that no such sequence exists. This is well known to be NP-complete, which makes exhaustive search of the circuit state space computationally impractical for large sequential circuits.

A well-known application of Boolean SAT is *Automatic Test Pattern Generation* (ATPG). ATPG is an approach that uses variants of DPLL[8] on a structural representation of the circuit. ATPG solutions have to satisfy the condition that a modeled fault on a line is *excited* by setting a value on the line, and *propagated*, by ensuring that the circuit with and without the fault differ by at least one Boolean output value. ATPG is used primarily for manufacturing-fault test-vector generation, but has found applications in logic synthesis and bounded model checking [3, 12]. The algorithms in exciting a fault correspond directly to the sequential SAT problem, and hence can be used to augment a traditional SAT engine.

Boolean SAT finds applications in many areas of circuit design and verification such as Bounded Model Checking [1], Redundancy Identification, and Equivalence Checking [15]. State-of-the-art SAT algorithms, as implemented in tools such as ZCHAFF [13] , have demonstrated that very hard SAT problems can now be solved in reasonable time. Bounded Sequential Search using SAT has been shown to be very effective in *model checking*. However, its biggest weakness is that of completeness in general sequential search. We demonstrate that ATPG techniques can close the loop to create a sequential SAT solver that retains the efficiency of SAT, while being complete.

The trade-offs between ATPG and SAT are summarized in Table 1 and are described below:

Circuit Information: ATPG guides the search process using structural information. ATPG typically finds solutions to satisfiable problems faster than SAT. However, SAT performs much better than ATPG on hard SAT problems, due to conflict-based learning. SAT can do quite well on manufacturing test-pattern generation and logic analysis [15], once it has access to structural information.

Solution Size: SAT assigns more primary inputs than ATPG when finding a satisfying pattern. This is due to its decision strategy that assigns internal variables, that typically force binary values on state variables. This is a problem when it is used in a true-sequential approach, where a solution for a time-frame is used to bound the search in future time-frames.

Implication Complexity: Implementation complexity of implication procedures in SAT is relatively lower than in ATPG. Hence, implication procedures in ATPG are typically slower than the corresponding *Boolean Constraint Propagation* (BCP) in SAT.

Learning : Modern SAT algorithms have more powerful conflict-based learning than typical ATPG algorithms, which makes SAT faster than ATPG on hard combinational problems, especially in UN-SAT cases [9].

SATORI's Focus It would be of tremendous interest to combine the strengths of both these techniques to solve the general problem of sequential SAT. SATORI addresses this goal by combining key advantages of both ATPG and SAT for efficiency and completeness. The key components of SATORI are:

1. Efficient implications using SAT-style implications.
2. Use of the circuit structure to guide the decision strategy.
3. True sequential search on a single time-frame at a time.
4. Conflict-based sequential learning and back-jumping.
5. Efficient state caching using a SAT database.
6. Efficient sequential bounding with minimal state cubes and a unique state-space avoiding clause scheme.

Paper Outline We describe some of the background in SAT and ATPG relevant to SATORI in Section 2. We present the basic architecture and algorithms used in SATORI in Section 3. We also analyze the key components in SATORI in this Section. We describe relevant prior work in Section 4. In Section 5, we present experimental results on the ISCAS'89 benchmarks to demonstrate the speed and capacity performance of SATORI as compared to a state-of-the-art commercial engine and the model-checker, VIS, v1.4. Finally, we present our conclusions in Section 6.

2 Preliminaries

In this section, we describe some of the background on the basic SAT algorithm and on gate-level justification. The basic SAT algorithm determines the satisfiability of a given problem in *Conjunctive Normal Form* (CNF). Here, we briefly describe the key components of generic SAT algorithms using conflict-based learning to allow understanding of the algorithms that follow. Interested readers may refer to *e.g.* [11, 13] for a more detailed description.

Notation Given a finite set of variables, $\mathcal{V}$, over the set of Boolean values $\mathbf{B} \in \{0,1\}$, a *literal*, $l/\bar{l}$ is a variable, $v/\neg v \in \mathcal{V}$. A *clause* c_i, is a disjunction of literals $(l_1 \vee l_2 \ldots, \vee l_n)$. A formula f, is a conjunction of clauses $c_1 \wedge c_2 \ldots, \wedge c_n$. A clause is considered as a *set* of literals and a formula, as a *set* of clauses. An assignment A for a formula f, is *satisfying* when the value of f given A is 1. An assignment is called *maximal* when its domain is $\mathcal{V}$. Following the convention in [13], we equate an assignment A with a disjunction of literals $v_i, \forall v \in domain(A)$, where $A(v_i) = 0$, and $\neg v_i, \forall v \in domain(A)$, where $A(v_i) = 1$. For example, the assignment $\{v_1(0), v_2(1)\} \equiv (v_1 \vee \overline{v_2})$. A *state clause* is a clause consisting only of state variables.

Boolean Constraint Propagation (BCP) The generic SAT algorithm decides on a sequence of variable assignments called *decisions*, which

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

	Feature	SAT	ATPG	Advantage
1	Conflict-Based learning	Yes	Minimal	SAT
2	Efficient Implications	Yes	No	SAT
3	Structural information	Some	Yes	ATPG
4	Decision Ordering	Appearance in clauses	Probabilistic	ATPG/SAT for sat./unsat. cases
5	Algorithm Complexity	Low	High	SAT
6	Size of SAT assignments	High	Low	ATPG

Table 1: SAT and ATPG feature comparison

generate additional variable assignments or *implications*. These implications are found by *Boolean Constraint Propagation*(BCP). BCP is a simple process for clauses since the only rule to be satisfied is that, given an assignment A, and a clause c_i in the formula, a variable v_j takes an implied value of 1 **iff**, all but one of the literals in the c_i are false.

Conflicts and Analysis 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 evaluating $f|_a$ is sufficient to cause the same conflict at v_i. A *conflict clause* is the clause corresponding to a. Given a CNF formula f and a conflict clause κ, then the new formula $f' \equiv f \wedge \kappa$, is satisfiable **iff**, f is satisfiable [11].

SAT Algorithm Termination Most SAT engines determine that a SAT instance is solved when the assignment A, is maximal and implication of these values can find no conflict [13]. Hence, by definition, a traditional SAT engine finds maximal assignments at state variables.

2.1 Structural Decision Strategies

ATPG engines differ in algorithms and data-structures, but they all share one common feature – they all explicitly use the structure of the circuit to guide the search process. The fault excitation part of an ATPG algorithm can be contrasted directly with general SAT solvers though it requires a 3-valued logic system {0,1,X}. Henceforth, we use the terms – sequential justification and sequential SAT interchangeably.

Back-Tracing Back-tracing is a structural circuit traversal method that selects a gate from a restricted subset of gates called the *justification frontier* or *J-frontier*. The signals in the justification frontier satisfy the property that the original justification objective is satisfied **iff** every signal in the justification frontier is satisfied. *Back-tracing* selects one of the gates in the *J-frontier* and recursively selects an unassigned input of the gate, using a 3-valued algebra $\in \{0,1,X\}$, in a depth-first manner until a valid decision point is reached [12]. Several heuristics exist in the literature [12] that can bias the assignment strategy toward the *primary inputs* (PIs), or toward state variables (also called *pseudo-primary inputs* (PPIs)), that can be more easily justified in time.

Circuit-level Implications BCP, or more commonly *implication*, for a Boolean gate is more complex than for a CNF formula. A *controlling* value c_g is an input value that uniquely determines the output value of a gate. For example, an **OR** gate has a controlling value of 1. The non-controlling value is $\overline{c_g}$. A gates *inversion parity*, $i_g = 1$, if the output of the gate is a negated primitive Boolean function. For example, **NAND, NOR** has an inversion parity of 1. The *output non-controlling* value, n_g^o, of a gate is $c_g \otimes i_g$. Similarly, the *output controlling* value, c_g^o, of a gate is $c_g \otimes i_g$.

The implication rules for a gate g are as follows:

1. An output is implied to a controlling/non-controlling value, if any of the inputs are at a *controlling value* or if all of the inputs are at non-controlling values.
2. All inputs are implied to *non-controlling* values if the output is at an *output non-controlling* value.
3. If the the output is at an *output controlling* value and all but one of the inputs are at non-controlling values, then the remaining input is implied to a *controlling* value.

As we can see, any gate-level implication routine would have to be substantially more complex than BCP on a CNF formula in order to support these rules over multiple primitive Boolean gates.

Structural Algorithm Termination An ATPG algorithm typically finds less than maximal assignments since it uses a 3-valued logic and path-based justification based on the concept of a *J-frontier*. However, there is no guarantee that these assignments will be *minimal*. Cheng *et al.*, [7] describes a procedure for finding minimal assignments in ATPG search. The algorithm requires search during the decision procedure and can be expensive, but is complete.

A *minimal* assignment is of great importance to a sequential SAT procedure, since every additional assignment on a state variable reduces the search space covered by the state vector by a factor of 2. Hence, *minimal* state variable requirements can help terminate a search quicker. We shall explore these concepts in greater detail with experimental analysis in Section 3.3.

In the following section, we describe how SATORI combines key concepts from ATPG and SAT to perform efficient sequential search.

3 SATORI

The key bottlenecks in a sequential SAT engine are: (a) The *implication engine*, (b) The *decision strategy*, and (c) The *state space pruning strategy*. We shall describe each of these components in SATORI and experimentally evaluate the efficiency of engineering the solver in this manner.

3.1 The Implication Engine

The implication engine in SATORI operates on a SAT-style data-base of clauses for speed. Consider a gate-level circuit $C \equiv \{S,G,L\}$, where S, is a set of state elements, G is a set of primitive Boolean gates, {AND, OR, NOT, NOR, NAND}, and L, is the set of signal lines interconnecting $g, g \in \{S \cup G\}$. Given C, we can convert it into a CNF formula f_C, whose variables correspond to each gate output $g \in G \cup S$ and whose clauses preserve the functionality of the gate. There is a one-to-one mapping from a value assignment and its implications, A_g, I_g in the circuit C, and the corresponding A_v, I_v in the formula f_C.

The efficiency of implications is significantly higher in a CNF formulation than in a gate-level circuit though the number of clauses is greater than the number of lines in the circuit. The main cost of the implication engine comes from evaluating when a gate is ready for implication. A SAT engine using *watched*-literals [13] can do this very efficiently. However, a gate-level implementation of watched-literals is not as efficient since the update of the watched-literals is complicated by the multiple implication rules for a gate level circuit.

3.2 The Decision Engine

The decision engine makes decisions on primary inputs and state variables for a user-defined number of decisions using back-tracing. The choice of decision variable is guided by sequential controllability measures [12]. If a satisfying assignment cannot be found by the cut-off number of decisions, the decision strategy changes to a VSIDS [13]-style approach, which weights the decision variables based on the number of clauses in which a literal appears, on an exponentially decaying scale over time. At this stage, all gates are considered as potential decision points. SATORI incorporates conflict-based learning [13] and back-jumping for efficiency in both the combinational and sequential search space. The interested reader is referred to [11] for more details of conflict-based learning and back-jumping.

Once a satisfying assignment for an objective in a time-frame is found, the assignments at the independent variables *i.e.,* PIs and PPIs are used to form a conflict clause, which represents the state that is required to satisfy the objective in the given time-frame. This *state clause* is used to prune the sequential search space as described in the following section.

3.3 State Space Pruning

In this section, we describe the algorithms in SATORI that allow us to perform sequential SAT efficiently. We describe a novel algorithm to solve the problem of minimizing the set of assignments made to state variables in a time-frame so that we can find a minimal state-clause for justification. We also describe how we use a clause-based mechanism for storing visited states and for pruning the sequential search space.

3.3.1 Assignment Reduction Algorithm

We now describe REDUCEASSIGNMENTS, which is a heuristic based on list propagation from the set of assignments on state variables that satisfies the given state objective for a time-frame. The algorithm takes as inputs a topologically sorted circuit *Ckt*, an assignment vector *A*, and a set of objectives obj; and produces a new assignment *a*, given by:

$$a \subseteq A \text{ s.t. } f_c(a) \wedge obj = 1, \textbf{iff } f_c(A) \wedge obj = 1 \quad (1)$$

REDUCEASSIGNMENTS is conceptually equivalent to existentially quantifying out the state variables in the formula $f_c \wedge obj$, one by one, in a given time-frame. In practice, the algorithm parallelizes this by propagating sets of *necessary* assignments. If there is no initial state specified, all the state variables in an assignment *A* can be quantified out only if *a* does not have any necessary state-variable assignments. Given a satisfying assignment *A*, there can exist several subsets $a_1, \ldots, a_n \in A$, with different sets of state variables quantified out, and each of which satisfy the objective in the given time-frame.

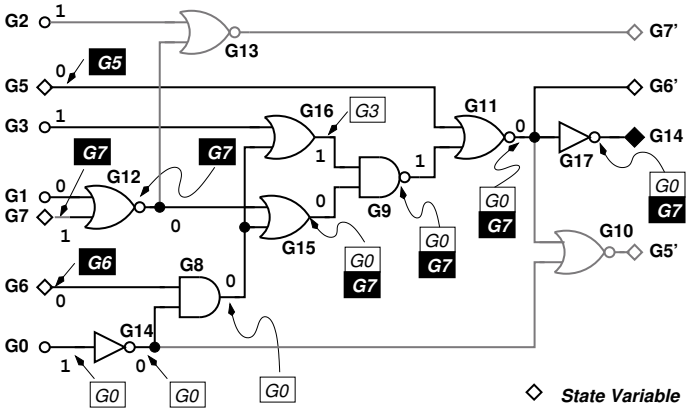

Figure 1: Example of REDUCEASSIGNMENTS.

If the value on the gate is a controlling value, then the PI/PPI that uniquely sets the gate value is chosen irrespective of whether it is a PI or a PPI. On the other hand, if there exists a choice between two controlling input variables, then we pick the first assigned PI with a controlling value in the lists that has been propagated to the inputs of the gate. If the gate is at a non-controlling value, then the lists at the inputs of the gate are merged at the output of the gate. Gates that are an unknown or X value are ignored. The final list at the justification objective gives the desired *minimal assignment set*.

This is illustrated in the Figure 1. The justification objective is $G14 = 1$. Assume that the decision procedure sets the values of $\{G0, G1, G2, G3, G5, G6, G7\} = \{1011001\}$, which satisfies the objective. 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 $G0, G7$. The time complexity of the algorithm is $O(VE)$, where V is the number of gates and E is the number of signal lines in the input cone of the justification objective. The space complexity is $O(VEk)$, where k is the sum of the PIs and state variables in the input cone, since we might in the worst case, have to propagate lists of all the PIs and state variables.

3.3.2 State Clause learning

SATORI uses conflict clauses to record the state requirements for a given time-frame. Given an assignment A_i that satisfies the objectives for time-frame t_i, the recorded clause is

$$c_i \equiv \text{REDUCEASSIGNMENTS}(A_i) \setminus \{v_i \in \text{PIs}\} \quad (2)$$

This corresponds to a set of states, in the state space due to X-values in $a \equiv \text{REDUCEASSIGNMENTS}(A_i)$, which cover several states at once.

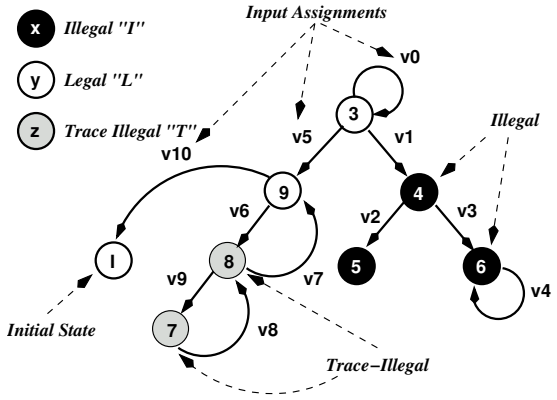

Figure 2: Different Types of State Clauses

The state clauses is stored in a state cache. The state cache can be conceptualized as a graph representing the state sets visited so far as shown as shown in the Figure 2. They can be classified as follows:

1. *Legal* states are states, $l_i \in L$, which can be reached from the specified initial state or from all states, if no initial state is specified (*e.g.* **9** in Figure 2).
2. *Illegal* states are states, $i_i \in I$, that are proved combinationally unsatisfiable in a time-frame i. This implies that they cannot be reached from any other legal state (*e.g.* **5,6**), or if they can be reached only from states which are truly combinationally unsatisfiable (*e.g.* **4**).
3. *Trace-illegal* states are states, $t_i \in T$, that are reachable only from states already visited and illegal states, *i.e.,* it is in a strongly connected component (SCC) and the trace has explored the SCC, without finding a solution.(*e.g.* **8**).
4. *Possibly legal* states are states, $x_i \in X$, that are visited in time-frame i, but not classified as illegal or trace-illegal. (*e.g.* **3**).

If the problem is satisfiable in a given time-frame, SATORI reduces unnecessary assignments on the state variables using REDUCEASSIGNMENTS and does a fast lookup to check if the state requirement covers a previously visited state. The results of the lookup are used to determine if:

- the decision procedure should terminate, or
- backtrack to an earlier time-frame, or
- backtrack in the current time-frame and continue the search.

We keep only one copy of the circuit in memory at a time. If we continue search in time, the formula f_{i+1} for the next time-frame is generated from the current f_i, and the learned state clauses $\{T, I\}$ as follows:

$$f_{i+1} \equiv f_i \wedge \{t_i \in T\} \wedge \{i_i \in I\} \quad (3)$$

In practice, this amounts to adding a few clauses to the current net-list. Identified *Illegal* states, which cannot be justified further back in time, are avoided implicitly from the formulation f_{i+1}.

Backtrack Clauses If we backtrack to an earlier time-frame, we create a *back-track* clause b_i corresponding to the literals in the last state clause learned in that time-frame and the value assignments on the PIs. Hence $f_i' \equiv f_i \wedge b_i$. f_i' creates the constraint that the last satisfying assignment cannot be repeated. Conflict analysis produces a conflict-avoiding clause which is used to maintain the decision tree in the time-frame i and continue search for the next satisfying assignment. Any

newly found illegal state clauses are added to f_i' to further prune the search space.

A back-track clause corresponds to a *cut* in the circuit with value assignments that guarantee that all decisions that imply or decide those value assignments will lead to a solution that has already been found. This approach differs from prior art in two key points:

- Earlier approaches [6] relied on using an expensive, lookup-table based scheme to avoid revisiting decision spaces. SATORI's clause-based scheme saves the cost of lookup and is correspondingly more efficient.
- [6] used the state of the *search-frontier* as a key for the lookup. SATORI's clause-based mechanism predicates search-bounding only on a combination of *value assignments*, which have been made before.

A typical DPLL back-tracking procedure will backtrack to the last decision made in the decision tree in a given time-frame and continue search. However, 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, with a minimum of conflicts in the decision procedure.

3.3.3 Loop Detection

SATORI performs DFS-search in the sequential state space. Hence, it is necessary to check for revisited states in order to avoid getting stuck in sequential loops. It would be correct to add the learned state clauses that are not classified as illegal in order to avoid loops.

The problem with this formulation is that these state clauses may imply values on state variables. Experiments showed that this typically takes the search on long state paths and significantly degraded performance. back-tracing typically finds much smaller paths, since we have controllability metrics to guide the search. However, this information is lost if we allow the learned state clauses to imply. This would not be the case in *image computation* [14], since that corresponds to a breadth-first search in the state-space.

Therefore, we check for loops by explicitly checking a new state against the set of visited states. This is done by comparing the literals in the new state versus the literals in the states in the cache. This check is actually quite fast, since we hash the states based on the literal values in a hash-table. A unique key is generated for each clause and is used to compare or retrieve visited states. We also check for Boolean covers in the visited states. If a state assignment is covered by an illegal state, then we can backtrack to the earliest time-frame where we detected the illegal state.

We also perform static illegal state identification, where state-cubes which are reachable only from illegal states are classified as illegal, and identification of legal state-cubes where known detection sequences exist. This is different from extracting a complete state-transition graph (STG), which can be quite expensive.

3.4 Termination

Given a satisfying assignment A_i, in a time-frame, SATORI terminates with:

1. SAT, if $a \equiv \text{REDUCEASSIGNMENTS}(A_i) \setminus \{v_i \in \text{PIs}\} = \emptyset$, *i.e.,* no state objective needs to be satisfied, or
2. SAT, if $a \subseteq S$, *i.e., a* is covered by a specified initial state S,
3. UN-SAT, if search on f' returns UN-SAT in the first time-frame *i.e.,* the sequential space has been completely explored.
4. ABORTED, if any of the user-defined parameters are exceeded.

3.5 Analysis of SATORI – Putting it all Together

In this section, we analyze some of the techniques described in the previous section with experiments on a subset of benchmarks from the ISCAS'89 benchmarks. The experiments are run on a small set of mid-sized circuits, to demonstrate the various aspects of SATORI . A full set of experiments on some very hard problems are shown in Section 5.

3.5.1 Decision Strategies

In this section, we analyze the decision strategies used in SATORI and their impact on sequential search.

Experiments were conducted on the sample benchmarks in order to illustrate the distinction between back-tracing (described earlier in Section 2.1) and a good SAT style decision strategy, VSIDS [13] (described in Section 3.2). Objectives of a Boolean 0 and 1 were set on each signal-line in each benchmark circuit, and checked one-by-one using the 2 competing strategies. Almost all the objectives are satisfiable, though with extensive search. The main points analyzed are:

1. How does back-tracing compare with VSIDS on time-frame expanded sequential circuits? How does true sequential search compare with search on time-frame expanded circuits?
2. How does back-tracing compare with VSIDS when used in true sequential search?

Ckts	R_d	ZCHAFF	SATORI	
			TF+Back-trace	True Seq
s1238	4	5.79	1.29	0.49
s1488	13	75.37	28.7	1.69
s1494	21	245.02	28.99	1.64
s344	10	1.94	0.59	0.15
s349	10	2.6	0.59	0.13
s386	9	3.37	2.22	0.13
s420	19	36.39	4.93	0.21
s510	11	18.32	11.91	2.03
s820	17	38.63	6.14	0.33
s838	19	221.98	19.67	0.83
s953	21	116.52	43.37	17.21

Table 2: Sequential Search v/s Bounded SAT

Back-Tracing on Unfolded Circuits In this set of experiments, we checked satisfiability on a sequential circuit as a combinational problem, and compared it to true sequential search.

The *recurrence diameter* R_d [1] of a sequential circuit, is defined as the minimum path-length including cycles from the initial state to the state where the objective is true. The sequential search problem can be modeled as combinational search on the sequential circuit unfolded R_d times. We estimated R_d, to be the maximum path length taken by SATORI in true sequential mode for each objective. While this is not a tight bound, it is a good approximation.

We compared the results of SATORI versus ZCHAFF. SATORI was run in 2 modes : (a) back-tracing on the unfolded circuit, and (b) True sequential search on the original circuit. ZCHAFF, was run on the unfolded circuit for each objective and the cumulative run-time for all objectives was calculated. The satisfying solution was checked for having no state variable requirements for correctness. The results are shown in Table 2. The Column R_d shows the number of time-frames for the unfolding for each circuit. The run-times for ZCHAFF on the unfolded circuit is shown in Column 2, and the run-times for SATORI for back-tracing on the unfolded circuit is shown in Column 3. Column 4 shows the run-times for full sequential search.

As we can see, back-tracing has a speedup of 2x–10x compared to ZCHAFF on all the circuits, while full sequential search performed from 10x–100x better than ZCHAFF, and again from 3x–20x better than back-tracing on the unfolded circuit. This clearly shows that decisions on the *independent* variables are better than *dependent* variables for satisfiable SAT problems. It has been shown, however, that the converse is true for hard unsatisfiable problems [9].

In the next set of experiments, we explore the performance of VSIDS versus a back-tracing strategy in full sequential search.

Back-Tracing on True Sequential Search VSIDS and back-tracing produce different state cubes to be justified during sequential search. We ran experiments on the sample benchmarks, with Boolean objectives of 0 and 1 on each signal line, and ran full sequential search on the circuits. We collected data on the number of state-cubes, which correspond to the number of time-frames searched. State caching was disabled to stress the two techniques in isolation.

The data in Table 3 show the results of these experiments. Columns 2 and 3 show the number of state cubes or time-frames searched for each strategy and Columns 4 and 5 show the respective run-times. The results

	State Cubes		CPU Time (secs)		
circuit	Struct.	VSIDS	Struct.	VSIDS	%change
s298	2400	2828	0.55	0.82	+49
s344	2117	57413	0.55	19.54	+3452
s349	2129	42112	0.67	15.77	+2253
s382	33331	209096	20.45	142.07	+594
s420	1769	2414	0.37	0.98	+ 164
s510	36699	39465	31.53	39.13	+24
s820	2737	7900	4.12	15.81	+283
s832	2854	7398	4.35	14.71	+238
s953	35154	53479	68.29	118.78	+73
s1238	187	285	0.6	2.41	+301
s1488	6452	19637	18.9	80.77	+327
s1494	7946	20685	28.19	84.48	+199

Table 3: Structural back-tracing v/s VSIDS [13]

clearly show that the number of state cubes in Column 3 of Table 3 that had to be justified for the case of the SAT strategy is significantly higher.

This behavior is due to the fact that the VSIDS strategy does not assign values based on any controllability metrics or structural guidance to minimize the number of assignments. A given objective can be satisfied with a variety of assignments. The best assignment would be one that makes a small set of controlling assignments so that the state cube is minimal. However, we found empirically that VSIDS makes a larger number of non-controlling assignments at the state variables. Hence, even the reduced state cube from VSIDS is still large as compared to the back-tracing method.

A state cube that has fewer unspecified values, and thus a smaller set of minterms typically generates the same kind of state cubes in the previous time-frames. Hence, the number of time-frames explored increases. This means that even with a heuristic to remove unnecessary assignments, the SAT strategy took longer state traces and found smaller covers than the back-tracing strategy. This directly translates to more run-time in the order of 1.5x–30x as shown in Column 6, since significantly more time-frames are explored than is necessary to find a solution. Most of the objectives in the above experiments have solutions and are found by both strategies.

3.5.2 REDUCEASSIGNMENTS

REDUCEASSIGNMENTS, which was described in Section 3.3.1, was used to remove the *unnecessary* assignments in each state cube. This is necessary since (a) the termination condition for the search is having no state requirements on the state variables, and (b) we wish to find state cubes with a maximum number of unspecified values in each time-frame, so that we can improve the search procedure. The SAT strategy produces a maximal assignment, as compared to back-tracing. Hence, we can safely conclude that REDUCEASSIGNMENTS is a necessity for a SAT style decision strategy. However, we compared SATORI with and without REDUCEASSIGNMENTS, when using the back-tracing decision strategy to determine the effects of assignment reduction.

Ckt	State Cubes		Max Path Len.		Run-Time(secs)	
	No-Red	Red	No-Red	Red	No-Red	Red
s1488	5831	259	31	13	14.27	1.69
s1494	8930	249	21	21	26.28	1.64
s382	50857	4601	750	279	27.58	19.3
s444	62095	4987	790	453	136.06	14.5
s510	65911	248	50	11	48.07	2.03
s820	4280	95	17	17	6.59	0.33
s832	4311	101	17	17	7.28	0.31

Table 4: REDUCEASSIGNMENTS and Back-Tracing

The experiments were again a set of objectives of Boolean 1 and 0 on each signal in the circuit. The results are shown in Table 4. Columns 2 and 3 show the number of state-cubes generated with and without REDUCEASSIGNMENTS, Columns 4 and 5 show the maximum length of the state trace taken during search, and Columns 6 and 7 show the respective run-times.

As we can see, the number of time-frames explored is significantly smaller with REDUCEASSIGNMENTS. The reduction is in the order of 3x–300x, with a corresponding reduction in run-time. The maximum state-trace that some of the examples took increased by upto 5x. It is clear that minimal state cubes can bound the search space significantly, and that REDUCEASSIGNMENTS is a useful heuristic in minimizing state requirement cubes.

3.5.3 State Caching

State caching is a useful technique for reusing the results of prior state traces. We ran some experiments on a few circuits, with state-caching turned on and off. The results are shown in Table 5. Columns 2 and 3 show the number of time-frames or state cubes explored without and with caching. Columns 4 and 5 show the corresponding run-times.

Ckt	State Cubes		Mem	Run-Time(secs)	
	no cache	cache	(KB)	no cache	cache
s1488	5397	259	7	18.4	1.69
s1494	7566	249	7	31.7	1.64
s382	171861	4601	293	262.47	19.3
s444	131343	4987	230	280.24	14.5
s510	12735	248	10	17.98	2.03
s526	111551	8327	458	454.35	66.68
s526n	108342	8208	456	459.85	69.7
s820	4624	95	5	10.89	0.33
s832	4637	101	5	11.05	0.31
s953	116187	149	49	3718.3	17.21

Table 5: Effectiveness of State Caching

The primary considerations, as in any caching mechanism, is the amount of memory consumed, and effectiveness of caching. As we can see from the table, the memory for caching never exceeded 500Kb, while reducing run-times by 8x–200x. Hence, state caching in the sequential search is a cheap and effective method to cut down run-time, when checking a number of objectives.

4 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. However, they suffer from two significant drawbacks – a) They cannot systematically avoid re-entering previously explored decision spaces, and b) They are significantly slower on hard combinational problems than state-of-the-art SAT solvers implemented specifically for solving a problem in a single time-frame. The subject of these two problems has been the focus of intense activity in the *Sequential Automatic Test Pattern Generation* over the last 4 decades. Cheng *et al.*, [12] presents a good survey of the current work in traditional ATPG.

SAT based techniques and implication graph based techniques have been used to speed up the basic combinational part of the problem [4]. A variety of techniques have been proposed for illegal state identification and storage to prune the state space effectively. SEST [6] uses RTP combined with limited learning to speed up sequential search.

Huang[10] proposed using ATPG techniques for verification including combinational equivalence checking and property checking. Boppanna *et al.*, [5] and Abraham *et al.*, [2] discussed the use of ATPG for model checking using a commercial tool. The former was a brute-force approach, while the latter's approach solely targeted checking a state space for bound k, using a commercial ATPG engine. Parthasarathy [9] compared SAT and ATPG for assertion checking and concluded that they have similar performance on small problem instances, while ATPG has advantages dealing with real-world circuits over SAT.

McMillan [14] proposed a method for quantification of variables in a CNF formula. This was targeted for image computation and is quite different from REDUCEASSIGNMENTS, since a) we perform DFS-like search in the sequential space and b) we perform quantification of subsets of state-variables in a time-frame. The approach by Cheng *et al.*, [7], performs search for minimal assignments, which can be quite expensive, while ours is a linear-time algorithm.

5 Experiments

So far, we have shown results on small to mid-sized circuits. However, we now show experiments designed to a) demonstrate the capacity performance of SATORI as compared to a state-of-the-art commercial solver, b) demonstrate the speed of SATORI on a set of justification objectives on hard examples with large state spaces, and c) demonstrate the power of sequential search when compared to a state-of-the-art BDD-based model checker – namely VIS, version 1.4 [16]

One interesting application of SATORI is in assertion checking on sequential circuits. Typically, we have a trade-off between BDD-based approaches and search-based approaches like *Bounded Model Checking*. Since SATORI is a complete search engine, we contrast it with VIS, on assertion checking on some large examples in the ITC'99 and ISCAS'89 benchmark suites. We ran 2 assertions on each example, which were:

1. $AG(p_1)$, where $p_1 = ((s_1 = 0) \vee (s_2 = 0), \ldots, \vee(s_n = 0)), s_i \in S$, where S, is the set of state variables in the circuit.
2. $AG(p_2)$, where $p_2 = ((s_1 = 1) \wedge (s_2 = 1), \ldots, \wedge(s_n = 1)), s_i \in S$.

SATORI was also compared to a commercial ATPG engine running in pure justification mode, which does not assume a complex fault model in these experiments, so that we have a fair comparison. The limits on SATORI and the commercial ATPG engine were a million back-tracks, 300 cycles and 3600 secs per property. VIS was given a time-out limit of 3600 secs, but no limits on memory. We used the standard VIS scripts, to perform the assertion checking. The experiments were run on an Intel Pentium-4 machine with 1 GB 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.

Ckt	SATORI	Rslt[2]	Comm. ATPG	Rslt	VIS	Rslt
b04	349.38	T/T	>1hr	T/A	>1hr	A/A
b11	1.12	T/T	0.7	T/T	3.1	T/T
b14	0.07	T/T	3.3	T/T	>1hr	A/A
b17	3.5	T/T	9.8	T/T	>1hr	A/A
b20	0.15	T/T	3.1	T/T	>1hr	A/A
b21	0.16	T/T	3.2	T/T	>1hr	A/A
b22	0.23	T/T	4.9	T/T	>1hr	A/A
s13207	1.2	T/F	2.8	T/F	>1hr	A/A
s15850	0.15	T/F	2.6	T/F	>1hr	A/A
s35932	119.5	T/T	>1hr	A/A	–	–
s38417	0.77	T/F	8.6	T/F	>1hr	A/A
s38584	0.41	T/F	7.0	T/F	>1hr	A/A
s5378	0.02	T/F	1.1	T/F	>1hr	A/A
s820	0.09	F/F	5.7	F/F	0.1	F/F
s832	0.09	F/F	5.6	F/F	0.1	F/F
s838	0	F/T	0.5	F/T	0.2	F/T
s9234	0.04	T/F	1.4	T/F	>1hr	A/A

[1] Run-Time in CPU seconds unless noted otherwise
[2]Status of $p_1/p_2 \in$ {T=proved true, F=proved false, A=abort}

Table 6: Assertion Checking Experiments

Column 2 and 3 show the results for SATORI , 4 and 5 for the commercial ATPG tool, and 6 and 7 for VIS. Most of these assertions required significant search in the sequential space. We show results only on the largest of the ITC'99 and ISCAS'89 benchmarks. All the engines finished quickly in under 0.5 seconds on all the smaller benchmarks. VIS aborted on most of the big benchmarks after about 3600 secs. The peak memory usage on these cases was close to 800 Mb. The memory consumed by SATORI did not exceed 150MB and the commercial engine did not exceed 300MB.

In the cases where VIS managed to finish building the BDD-based images, it outperformed the commercial engine by 2x to 50x. However, SATORI finished all the test-cases including cases where both the commercial engine and VIS failed in reasonable time. The commercial ATPG engine also finished in all but 2 cases, with very reasonable times. This clearly demonstrates the power of sequential search for assertion checking. SATORI outperformed commercial ATPG by 20x to 50x and finished on 2 cases where ATPG aborted after 3600 secs, except in **b11**.

6 Conclusion

We have described SATORI – a general sequential justification engine that is at least an order of magnitude faster than a state-of-the-art commercial tool. SATORI addresses limitations of traditional SAT solvers and structural sequential ATPG-based satisfiability. We have described how we have integrated path-based justification with conflict based learning and SAT techniques in SATORI for efficient sequential search. Our analysis shows that achieving minimal state requirements in each time-frame using back-tracing and heuristic assignment reduction improves efficiency significantly. We have also presented sample results that demonstrate how the strengths of each technique can be effectively integrated into an efficient solver. We have presented results on SATORI versus VIS, which show that SATORI can be more efficient than BDD-based image computation when exploring huge state spaces. We have also shown that it is often better to use a complete sequential search engine than on a naive bounded model.

Future improvements includes strengthening the sequential learning and structural search, and definition and implementation of the full semantics of a temporal logic in SATORI for symbolic model checking.

7 References

[1] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y.Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proc. of the 36th ACM/IEEE DAC*, pages 317–320, June 1999.

[2] J. Abraham and V. Vedula. Verifying Properties Using Sequential ATPG. In *Proc. of ITC*, pages 315–319, Oct. 2002.

[3] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. CS Press, 1^st ed., 1990.

[4] A. Biere and W. Kunz. SAT and ATPG: Boolean Engines for Formal Hardware Verification. In *Proc. of the IEEE/ACM ICCAD*, November 2002.

[5] V. Boppana, S. Rajan, K. Takayama, and M. Fujita. Model Checking Based on Sequential ATPG. In *Proc. of Int. CAV'99*, LNCS, pages 418–430. Springer, July 1999.

[6] X. Chen and M. Bushnell. Sequential Circuit Test Generation Using Dynamic Justification Equivalence. *JETTA*, 8(1):9–34, Feb. 1996.

[7] K. Cheng and H. Ma. On the Over-Specification Problem in Sequential ATPG Algorithms. In *Proc. of the* 29^{th} *DAC*, pages 16–21, June 1992.

[8] M. Davis, G. Logemann, and D.Loveland. A Machine Program for Theorem Proving. *Journal of the ACM*, 5(7):201–215, 1962.

[9] G. Parthasarathy, C.-Y. Huang, and K.-T. Cheng. An Analysis of ATPG and SAT Algorithms for Formal Verification. In *Int. Workshop on HLDVT*, pages 177–182, November 2001.

[10] C.-Y. Huang, B. Yang, H.-C. Tsai, and K.-T. Cheng. Static Property Checking Using ATPG v.s. BDD Techniques. In *Proc. of ITC*, Aug. 2000.

[11] J.P Marques-Silva and K.A. Sakallah. GRASP - A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, May 1999.

[12] K.-T. Cheng and A. Krstic. Current Directions in Automatic Test-Pattern Generation. *IEEE Computer*, 32(11):58–64, 1999.

[13] M. Moskiewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an Efficient SAT Solver. In *Proc. of the 38th ACM/IEEE DAC*, 2001.

[14] K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proc. of the* 14^{th} *CAV*, volume 2402 of *LNCS*, pages 250–264. Springer-Verlag, 2002.

[15] P. Tafertshofer, A. Ganz, and K. Antreich. IGRAINE – An Implication-Graph based Engine for Fast Implication, Justification and Propagation. *IEEE Trans. on Computer-Aided Design*, 19(8):907–927, August 2000.

[16] R. K. Brayton, G. D. Hachtel, *et al.*, VIS: A System for Verification and Synthesis. In *Proc. of the* 8^{th} *CAV*, volume 1102 of *LNCS*, pages 428–432, July/August 1996. Springer Verlag.