Automatic Lighthouse Generation for Directed State Space Search

Praveen Yalagandula
ECE Department
The University of Texas
Austin, TX

Vigyan Singhal
Tempus Fugit, Inc.
525 Curtis St
Albany, CA 94706

Adnan Aziz
ECE Department
The University of Texas
Austin, TX

Abstract

Previous researchers have suggested the use of "lighthouses" to act as guides in directed state space search. The drawback of using lighthouses is that the user has to manually derive them, through a potentially laborious examination of the design. Additionally, specifying a large number of lighthouses results in wasted effort during the search. We present approaches to automatically generate high-quality lighthouses for hard-to-cover targets.

1 Introduction

We address the problem of efficiently searching the state space of synchronous digital hardware designs, starting from a single designated initial state. State space search has many applications — verification of safety properties (e.g., bus contention), justifying the reachability of a given state for an equivalence checker, generating tests to satisfy missing coverage, analysis of third–party RTL code to understand behavior, etc.

Conventionally, state space search has been performed by *simulation*. Large numbers of input sequences, called tests, are applied to a software model of the design; these tests are generated by random test pattern generators, or by hand. Simulation is simple, and scales well in the sense that the time taken to simulate is proportional to the design size. However, for large designs, the fraction of the state space which can be covered in this methodology is vanishingly small [1].

This state of affairs has led to the proposal of *symbolic* search strategies, based on the use of BDD's to implicitly represent set of states, next state functions, and perform basic FSM manipulations [4]. Conceptually, these approaches systematically explore all states reachable from the initial state. The computational complexity of symbolic search is enormous; as such it is limited to designs containing of the order of a hundred latches.

1.1 Directed Search: SIVA

Ganai *et al.* [2] present a stand-alone tool, SIVA, which combines simulation with symbolic methods to form a robust method for state space search directed towards user-specified *targets*.

The working of SIVA is described as follows. The design is read into SIVA as a netlist of gates and latches. Targets are specified $(node, value)$ pairs, where $node$ is a node in the design, and $value$ is 0 or 1. SIVA then tries to find input sequences which lead to *covering* the targets, i.e., for each target, set its $node$ to the corresponding $value$. The tool starts by simulating randomly generated input vectors at the initial state; this gives a set of states that can be reached in from the initial state. The routine *randomSimulate* performs this step and returns the new states reached. This function also prunes the target list T if any targets are satisfied in simulation. The tool stops if all targets are covered in random simulation. Otherwise, a deterministic procedure, *symbolicSolve* is invoked.

The function *symbolicSolve* takes as arguments a state s and a target $aTarget$. It attempts to generate an input vector which on application at s leads to a state where $aTarget$ is covered; if it is successful, *symbolicSolve* returns the sequence (called a *witness*). The *symbolicSolve* uses a combination of SAT-based ATPG [5] and BDD building with size and backtrack limits to keep it robust. SIVA stops if all targets are reached by invoking *symbolicSolve*. Otherwise, the tool selects a state from the reached states and starts exploring the next states by random simulation. This process continues till either all targets are satisfied or the computational resources are exhausted.

Generally, targets which are difficult to cover with SIVA are distant from the initial state in the state transition graph. Purely simulation based search is prone to getting lost in the state space, exploring states from which target cannot be ever covered.

It is often the case that the user may be able to give "hints" which will help generate inputs leading to the targets. For example, if a target is several levels of conditionals deep, it is necessary to satisfy the preceding conditionals before the target can be satisfied. Users can specify such hints

to SIVA in the form of *lighthouses*. A lighthouse is also specified as a $(node, value)$ pair, covering which is taken by SIVA as evidence that the search is proceeding in the right direction. Operationally, when lighthouses are specified, the *symbolicSolve* routine tries to satisfy lighthouses along with targets.

An example of the use of lighthouses is given in Figure 1. In this design, u is of 8 bits length and a, b and c are single bit latches. The goal is to set the latch c to 1. In this case, it can be readily seen that $(a, 1)$ and $(b, 1)$ are possible lighthouses for the target $(c, 1)$. If we do not specify the lighthouses, then it is hard to find the input sequence which satisfy the target.

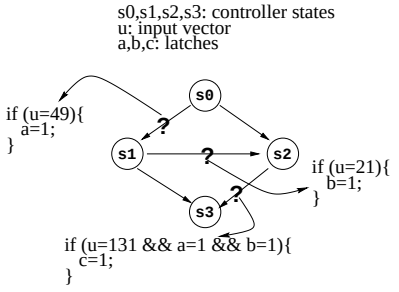

Figure 1. Control flow illustrating the need for lighthouses.

Ganai *et al.* [2] observed that lighthouses play a major role in generating input sequences for hard-to-cover targets. The drawback of using lighthouses is that the user has to manually examine the design to find them. This can be tedious, and takes away from the usefulness of SIVA. In addition, specifying an excessively large number of lighthouses results in performance degradation, since SIVA (as described in [2]), applies the *symbolicSolve* routine to each of them. In this paper, we develop approaches to automatically generating high-quality lighthouses for hard-to-cover targets.

2 Lighthouse Generation

2.1 Intuition

Conceptually, we want to automatically generate in a pre-processing step conditions on various nets of the design whose satisfaction during the search guarantees "progress" towards the target. Note that there is usually some ordering present between the conditions. So, in addition to lighthouses, we want to schedule them so that when we satisfy them in given order, we make progress towards the target. In this way, we avoid the inefficiency of trying to satisfy all of them simultaneously. For simplicity, we generate conditions only on latch outputs, rather than arbitrary nodes in the design.

A naive approach is to use every latch in the transitive fanin of the target as a lighthouse for that target. There are several problems with this approach:

1. The number of latches present in the transitive fanin cone of target is very large, making the *symbolicSolve* step in SIVA a performance bottleneck.

2. The graph may be cyclic, and hence will not tell us where to start.

3. For some targets, it may not be necessary to toggle a fanin latch to cover the target.

Again, since there is usually some ordering present between the conditions needed to cover a target, we need some way of ordering latches so that we concentrate on a subset at each symbolic solve step. We propose a way of finding such a partial ordering the latches by building a *latch graph*. Then during the search, we use only those latches as lighthouses, whose predecessors in the partial order are already satisfied.

Let us suppose that to set a latch l to value 1, we need another latch m to be 1 in previous cycle. If $(l, 1)$ is our target, then $(m, 1)$ is a natural lighthouse for the target $(l, 1)$. So the first step towards finding the schedule of lighthouses is to identify edges between the vertices which must be taken to progress from one vertex to the next. Since conditions which set a latch from 0 to 1 and conditions which set the latch from 1 to 0 are different, we have two different vertices for each latch in the latch graph.

Formally, the latch graph is defined on the vertex set $\{(l, b) \mid l \in L, b \in \{0, 1\}\}$, where L is the set of latches in the design. Initially, an edge $\big((l_1, a), (l_2, b)\big)$ exists in the latch graph if and only if a combinational path from l_1 to l_2 exists in the design. An edge from (v_1, p) to (v_2, q) in the latch graph is defined to be a *required* edge if the toggling of latch v_2 from $\overline{q}$ to q requires v_1 to be p in the previous cycle. Since at the initial state half of the vertices are satisfied, we concentrate only on the other half of the vertex set. In particular, the initial-valued vertices (vertices of the form (l, α) where α is the initial value of latch l) will not have any fanin vertices in the latch graph.

It may be that in the latch graph the target vertex have incoming edges which are not required edges. Suppose we have an edge from $(a, 1)$ to $(b, 1)$ in the graph. Also assume that there exists an assignment to the inputs of the fanin cone at the input of latch b in network which sets the output to 1 regardless of latch a's value. Heuristically, assigning $(a, 1)$ to be a lighthouse for $(b, 1)$ is not useful. So we remove such edges in the second step, by performing universal quantification on latches with respect to their fanin latches.

We now present an example of a design with five latches illustrating the significance of above two steps and the reason for their order of execution. Consider the RTL code fragment shown in Figure 2. Here a, b and c are single

```
assign a = (input==134) ? 1 : 0;

always@(posedge clk) begin
    if (a||b){
      c = 1;
      count = 2'b00;
    }
    else if (c==1){
      if (count==2'b11){
        c = 0;
        b = 1;
      }
      else{
        count ++;
      }
    }
end
```

Figure 2. RTL code fragment

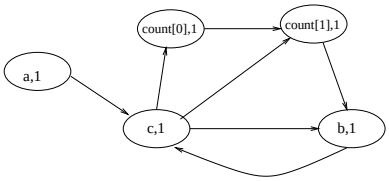

Figure 3. Latch graph for code shown in Figure 2

bit latches and $count$ is a 2-bit latch. Assume the target is $(b, 1)$. The initial latch graph will be as shown in Figure 3, assuming the initial values of all latches as 0. (For clarity, we do not show initial valued vertices in the figure). Consider the vertex $(c, 1)$. If we perform universal quantification step on this vertex, both the edges from $(a, 1)$ and $(b, 1)$ are removed. But we know that to reach the target $(b, 1)$, we need to satisfy $(a, 1)$. So we apply following rule when finding required edges of a vertex:

Rule RA: If (l_p, r) is required to be satisfied before (l_q, s) can be reached, then assume l_q set at value $\overline{s}$ when finding required edges of vertex (l_p, r).

When we find required edges, the edge from $(c, 1)$ to $(b, 1)$ will turn out to be a required edge. So applying the rule, we can assume b to be 0 when we are finding the required edges of $(c, 1)$. This time the edge from $(a, 1)$ to $(c, 1)$ will become a required edge. We can remove the edge from $(b, 1)$ to $(c, 1)$ in this case. Once we find the required edge from $(a, 1)$ to $(c, 1)$, by Rule RA, we can assume that both b and c are 0 when we are trying to satisfy $(a, 1)$.

We have found that Rule RA is quite important and powerful in practice, finding many useful lighthouses. With each required edge found, say from (l, p) to (m, q), we may find more required edges for vertices in the transitive fanin of (m, q). So we need to call the required condition finding algorithms iteratively till no new edges are found.

Another useful observation is the fact that we can prune some edges when we find required edges: if we find that an edge from $(a, 1)$ to $(b, 1)$ is a required edge, then we can remove the edge $(a, 0)$ to $(b, 1)$. Some more edges can also be removed on basis of information from using Rule RA. For example, for the latch graph in Figure 3, we can remove edge from $(b, 1)$ to $(c, 1)$.

After completing the above two steps, we will have two types of edges left in the latch graph. Some of these edges are required condition edges found in first step; other edges correspond to those edges which did not get removed in universal quantification. Because of these edges the latch graph may not be a DAG.

At this stage, we form the graph of strongly connected components and treat each SCC as an entity to satisfy. Essentially, all vertices in an SCC σ_1 are treated as lighthouses for vertices in SCC σ_2 having σ_1 as a predecessor in SCC DAG. (Our experiments observes that most of the SCCs are of single vertex.) We maintain a frontier to keep information about how close to the target we have moved and apply the latches before the frontier as lighthouses. Initially, the frontier will have the initial valued leaves of the DAG rooted at target. The procedure for constructing the latch graph is described in detail in the next section.

2.2 Implementation

The routine *latchGraphConstruct* constructs the latch graph from the given network applying the two steps explained in previous section. Initially, this routine forms the graph (V, E), where $V = \{(l, b) \mid l \in \{\text{latches in network}\}, b \in \{0, 1\}\}$ and the edge set $E = \{((v_1, p), (v_2, q)) \mid v_1 \text{ is in transitive fanin of } v_2\}$. The second step of the procedure is to find required edges by the method of constant propagation, explained in Section 2.2.1. The method of constant propagation is incomplete — it does not find all required edges. To find the complete set, we use an approach based on ATPG and simulation, explained in Section 2.2.2. These two routines are invoked iteratively till no new required conditions are found. The last step is to perform universal quantification of all nodes with respect to their inputs, described in Section 2.2.3.

2.2.1 The Method of Constant Propagation

The first step in the procedure is to find required edges between vertices. Some required conditions can be easily found simply by propagating the value at the output backwards. We found it to be quite useful on many latches, though all required edges can not be found.

For each vertex in the graph, we extract the single output combinational cone whose output is the input to the latch corresponding to the vertex. The next step is to set the already known inputs which arise by applying *Rule RA*.

We do a forward propagation of known values before going for backward propagation of values. These techniques are quite well used in the ATPG field [3] to generate the input vector satisfying the given logic network.

2.2.2 ATPG and Simulation based methods

Constant propagation will not be able to find all required conditions for a vertex. We need to use some other methods for finding this type of required conditions. One approach might be to build BDDs for the cones, and perform cofactoring. However, building BDD's tends to be computationally infeasible. Another approach is to use ATPG techniques to resolve the dependencies by performing two ATPG calls per each fanin of a vertex (setting fanin to 0 once and 1 next time). A better way of doing the same is shown in Algorithm 1, which cuts down half of the above ATPGs at the cost of one ATPG for each vertex.

Here for each vertex $n = (l, p)$, we perform ATPG to check the whether $\mathcal{F}_l$, the function at the input of latch l, can be set to value p. We impose following constraints on the ATPG problem: *(a)* the required edges are to be satisfied, and *(b)* conditions because of *Rule RA*. These two conditions are always imposed on all ATPG queries invoked in the algorithms. If $\mathcal{F}_l$ is satisfiable, the ATPG tool returns the witness t_n. Latches corresponding to fanin vertex $f = (m, q)$ of n will have some value in t_n. We perform another ATPG operation to check if $\mathcal{F}_l$ is satisfiable when the value of m is toggled from its value in t_n. If it is still satisfiable, then we know that f is not a required fanin of n.

Algorithm 1 atpgReqdCondFinder

```
1:  for each vertex n do
2:    if faninToCheck(n) = 0 then
3:      continue
4:    L ⇐{fanins of n to check for}
5:    t_n ⇐ ATPG for n
6:    while L ≠ ∅ do
7:      Pick a fanin f ∈ L
8:      u ⇐ ATPG for n with f flipped
9:      if u = ∅ then
10:       (f, n) is a reqd edge.
```

Still, the number of ATPG operations performed on invocation of above algorithm is quite large — the number of vertices plus the number of edges. We can use the witness u found in Step 8 to reduce some ATPG operations as follows. Check for fanins whose values in t_n are different from u. We can safely remove them from the list L. Another improvement comes from the fact that the witness returned by ATPG contains some dont cares, i.e., some of the bits in the witness are X's. We can remove the corresponding fanins also from the list L. The ATPG tool, we used [5], does not always return all possible dont cares in the witness. We find the remaining dont cares in $simulateRemove1$ routine which is invoked by the algorithm on every witness found.

Another enhancement to the above procedure to further prune the number of ATPG operations comes from the observation that the combinational input cones to latches are not different for all latches. Most of them share a great number of gates. Since simulation is cheap step, we can use the previously found witnesses to check if they satisfy the present vertex under consideration. The final version of algorithm with all enhancements is shown in Algorithm 2.

Algorithm 2 atpgSimReqdCondFinder

```
1:  T ⇐ ∅
2:  for each vertex n do
3:    if faninToCheck(n) = 0 then
4:      continue
5:    L ⇐{fanins of n to check for}
6:    if any t_i ∈ T satisfies n then
7:      t_n = t_i
8:    else
9:      t_n ⇐ ATPG for n
10:     T ⇐ T ∪ {t_n}
11:   simulateRemove1(t_n, L)
12:   while |L| ≠ 0 do
13:     pick a fanin f ∈ L
14:     u ⇐ ATPG for n with f flipped
15:     if u then
16:       for all m ∈ L do
17:         if m has diff. values in t_n and u_fn then
18:           L ⇐ L − {m}
19:       simulateRemove1(u, L)
20:     else
21:       (m, n) is a reqd edge
```

2.2.3 Universal Quantification Step

The second step in the construction of latch graph is to remove some edges by performing universal quantification. We can implement this step either by using BDDs or using ATPG techniques. As mentioned earlier, the use of BDDs is limited to small networks — for big networks, construction of the BDD for the next state function for even a single latch can cause memory explosion.

Our approach is to duplicate the given combinational net and tie together the corresponding inputs in two nets except the inputs corresponding to variable, on which universal quantification has to be done. To check whether $\forall a(\mathcal{F}_b)$ is satisfiable, we set the input a in one net to 0 and in other net to 1. Then we feed this net to the ATPG solver and check for a stuck-at-0 fault.

The algorithm for this step is shown in Algorithm 3. We use some of the simulation techniques used in finding required edges here to speed up the step. The $simulateRemove2$ routine invoked from this step acts in a slightly different way compared to $simulateRemove1$ used in required edge finding algorithms. If we find that a latch b is a dont care in the witness of vertex (l, p), then

we remove the edges $(b, 0)$ to (l, p) and $(b, 1)$ to (l, p) from the latch graph.

The witnesses set T collected in previous step shown in Algorithm 2 is passed as an arguement to this procedure. We found it was quite useful in pruning lot of ATPGs, which otherwise would have been called in Step 8 of Algorithm 3.

Algorithm 3 univQuantification(witnessSet T)

```
1:  for each vertex n do
2:    if faninToCheck(n) = 0 then
3:      continue
4:    L ⇐{fanins of n to check for}
5:    if any t_i ∈ T satisfies n then
6:      t_n = t_i
7:    else
8:      t_n ⇐ ATPG for n
9:      T ⇐ T ∪ {t_n}
10:   simulateRemove2(t_n, L)
11:   while |L| ≠ 0 do
12:     Pick a fanin f ∈ L
13:     u ⇐ ATPG for ∀f(n)
14:     if u then
15:       simulateRemove2(u, L)
```

3 Experiments

In this section, we give an outline of the design used in our experiments, which is a data decompresser chip. `Decompressor` is the entire design. This module has 70 inputs, 10333 latches. After decomposing into NAND gates, the design has 109666 gates. `TreeDec` is a part of `Decompressor`. `TreeCtl` is a simple controller inside `TreeDec`. `TreeDec` and `TreeCtl` share some of the inputs supplied by the `Decompressor` module (ex: `BOB`). `TreeDec` has 49 inputs, 2864 latches and 38161 gates; `TreeCtl` has 26 inputs, 75 latches and 1161 gates.

`BOB` is an input signal to the `TreeCtl` module which starts the controller. The sequential flow of different steps in the `TreeCtl` module are shown in the Figure 4. The names inside ellipses corresponds to the latches in the design. Initially all latches are set to 0. A name in ellipse denotes that the corresponding latch is set, on reaching that step. And when the controller goes out of that step, the latch is reset. `OpInd` is a 2-bit register and its value is retained through the steps. `HCLen`, `HLit` and `HDist` are 6-bit registers and `SymCnt` is a 9-bit register.

The targets we have chosen for our experiments on `TreeDec` and `TreeCtl` are `CLTab`, `WAT`, `CodGen`, `GoBsCd` and `Done`. In addition, we chose `BOB` as a target while experimenting on `Decompressor`. The toughest target of all is `Done` which is located quite deep in the state space. `Done` can be 1 only when `DnCdGn` is 1 and `(OpInd==’00’)`. The first time `DnCdGn`

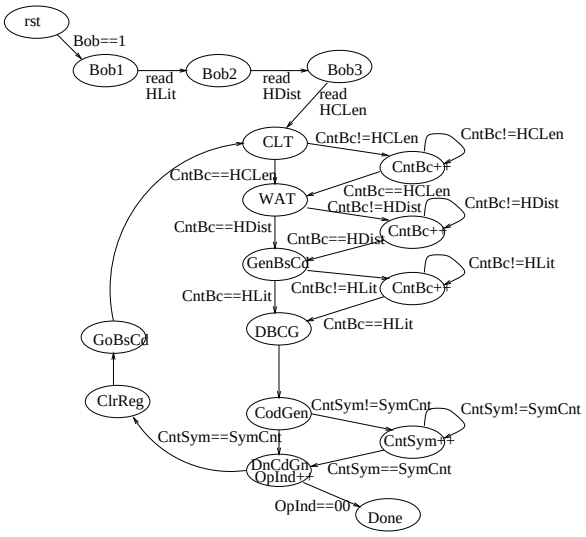

Figure 4. Flow diagram of TreeCtl module

becomes 1, `OpInd` increments to 01. Thus, the controller has to go through the loop three more times before `(OpInd==’00’)` is satisfied.

In the version of SIVA we were given access to, the state selection strategy was to pick one of the reached states randomly. We experimented with the strategy of picking a newly reached state instead of random selection; we call this version siva-ϵ. When applied on `TreeCtl`, all targets were reached even without the specification of light houses. But when applied on `TreeDec`, it could only reach till `CLTab`. This is due to the existence of many other paths from the starting states.

We experimented with the new SIVA algorithm enhanced with automatic light house generator. All the experiments were done on a 450MHz Pentium-II running Linux. The results of different steps performed on latch graph are tabulated in Table 1. All times are in seconds.

Constant propagation results are in rows whose heading is *const Prop*. From the results, it is easy to observe that a lot of required edges were found by this method. The row with heading *rem edges* gives the number of edges removed in this method. The remaining required edges are found using ATPG based techniques. Some edges are also removed in this routine. These are also presented in the Table 1.

The results of three different algorithms for finding required edges for the above mentioned designs are shown in Table 2. The *alg1* corresponds to Algorithm 1, *alg3* is the final version given in Algorithm 2 while *alg2* is just as Algorithm 2 without using the witnesses found in previous ATPG operations. The results show that the integration of the simulation techniques with ATPG techniques is quite efficient than using ATPGs alone. The results also show that the augmentation step used in final version is quite successful in pruning the number of ATPG operations.

The universal quantification step’s results are shown in

stats		TreeCtl	TreeDec	Decomp
Initial	vertices	102	962	4424
	edges	1276	37252	163028
Const Prop	reqd edges	192	1995	11539
	rem edges	387	2523	12408
	time(secs)	0.11	24.5	349
ATPG	reqd edges	33	68	734
	rem edges	71	144	854
Univ Quant	rem edges	548	31330	133556
	time	0.82	336	6097
	ATPG calls	43	764	6231
	ATPG time	0.70	291	5167
	Sim time	0.09	38	846
graph stats	vertices	102	962	4424
	edges	270	3255	16210
num sccs		101	961	4423

Table 1. Results of *latchGraphConstruct* on three designs. Decomp refers to Decompressor

Design		alg1	alg2	alg3
TreeCtl	time	1.87	1.05	0.9
	ATPG calls	210	129	102
	ATPG time	1.81	0.9	0.76
	Sim time	0	0.08	0.11
TreeDec	time	4823	248	212
	ATPG calls	4201	1171	958
	ATPG time	4816	217	169
	Sim time	0	24	34
Decompressor	time	>24hrs	4079	3675
	ATPG calls	n/a	7258	6555
	ATPG time	n/a	3529	2876
	Sim time	0	406	665

Table 2. Results of the experiments with different versions of required condition finding algorithms explained in Section 2.2.2

rows with heading *Univ Quant* in Table 1. Finally, the number of SCCs in the final graph are shown. Only one of the SCCs contained two vertices and all other SCCs were of single vertices.

The new SIVA was executed on `TreeCtl` and `TreeDec` with automatic light house generator. The tool was able to cover all targets in `TreeCtl`, while it reached the target `GoBsCd` in `TreeDec`. But it could not reach the target `Done`. The results of experiments on `TreeCtl` and `TreeDec` are tabulated in Table 3 and Table 4.

4 Conclusions

In summary, we proposed a heuristic to generate light houses automatically which will help the state space search tool in reaching the targets specified by the user. The results show that the modified SIVA with automatic light house generator was able to find more targets than the original SIVA.

This is part of an ongoing project. We have more ideas related to guided search that we are working on. Right now the latch graph is built statically as a preprocessing step. In future, we would like to update the latch graph dynamically depending on the states reached. Another idea is to make the graph where vertices will not be just corresponding to a particular latch state, but corresponds to the state of a particular group of latches.

Target	`CLTab`	`WAT`	`CodGen`	`GoBsCd`	`Done`
siva	yes	no	no	no	no
siva-ϵ	yes	yes	yes	yes	yes
siva-II	yes	yes	yes	yes	yes

Table 3. The performance of different versions of SIVA on TreeCtl design. 'yes' denotes that the target is reached. siva-ϵ is sivaModified without using light houses, siva-II is with light houses.

Target	`CLTab`	`WAT`	`CodGen`	`GoBsCd`	`Done`
siva	yes	no	no	no	no
siva-ϵ	yes	no	no	no	no
siva-II	yes	yes	yes	yes	no

Table 4. The performance of different versions of SIVA on TreeDec design. The notation is same as used in Table 3

References

[1] B. Chen, M. Yamazaki, and M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proc. European Conf. on Design Automation*, pages 132–136, March 1994.

[2] M. Ganai, A. Aziz, and A. Kuehlmann. Enhancing Simulation with BDDs and ATPG. In *Proc. of the Design Automation Conf.*, New Orleans, LA, June 1999.

[3] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C-31:215–222, 1981.

[4] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[5] J. Silva and K. Sakallah. GRASP–A New Search Algorithm For Satisfiability. In *Proc. Intl. Conf. on Computer-Aided Design*, Santa Clara, CA, November 1996.