

Improving Coverage Analysis and Test Generation for Large Designs

Jules P. Bergmann and Mark A. Horowitz
Computer Systems Laboratory,
Stanford University,
Stanford, CA 94305
(bergmann@cs.stanford.edu, horowitz@stanford.edu)

Abstract

State space techniques have proven to be useful for measuring and improving the coverage of test vectors that are used during functional validation via simulation. By comparing the state and edge coverage provided by tests with that which is possible in the design's state graph, the designer can estimate how well tested the design is and identify areas that need better testing. Unfortunately, for many interesting designs, the full state graph may be too large to fully explore, or if it is explorable, the resulting coverage may be so low as to provide limited feedback. Several techniques have been proposed that identify and work with an interesting subset of the design's state machines, but they still require computing the full state graph before projecting it. In this paper we discuss *projection directed state exploration*, in which a projection from the full graph is found while exploring only the relevant portion of the full graph. Even with this limited exploration, BDD size blowup is still a problem. To deal with this, we have also developed several interactive tools that provide feedback to the designer, and allow them to add hints to help with the exploration.

1 Introduction

Functional validation of designs is generally done by simulation of the design's RTL model. A range of methods exist to generate test vectors, but it is important that those vectors exercise all of the design's interesting behaviors to expose bugs. State and edge coverage of the design's state graph have been proposed as coverage metrics to evaluate how well the design has been tested [3, 6].

These methods work by exploring the design's state graph, typically with symbolic BDD techniques, and then comparing the total number of states and edges with those actually covered by simulation. Additionally, the design's state graph can be used to generate tests [4] to fill in gaps in the coverage.

State explosion - the exponential growth in the number of states as the design size increases - is a problem for these techniques. Abstraction of the design, either by extracting the control portion of the design (the datapath is amenable to other forms of testing), or by creating an abstract model of the design that models key features [2], must be done.

Even then, the abstracted state space may be too large to fully explore, or if it is explorable, the resulting coverage may be so low as to provide limited feedback. To address this, Ho [3] recognizes that control variables are not equally important in their effect on the design's behavior and defines the concept of a control event to identify an interesting subset of the control variables. Geist *et al* [2] rely on the designer to label the state variables as *cover*, *care*, or *ignore* to accomplish a similar reduction.

The common requirement of these techniques is the accurate *projection* of the design's large control graph onto a subset of interesting state variables. Moreover, they do not need the full control graph, only enough to make the projection and allow translation of path in the projection graph to concrete paths in the real graph.

In this paper, we will discuss the technique of *projection directed state exploration*, in which a projection from a full graph is found while exploring only the portion of the full graph relevant to the projection. This technique is based on standard symbolic BDD methods and has been used

to extract projections of large control graphs that would otherwise be intractable.

As with other techniques, state explosion and BDD size blowup are problems that must be dealt with. We present several techniques to delay the onset of size blowup by reducing the state space being explored and by focusing on doing easier work first. For large enough designs, size blowup is inevitable. Since our goal is to provide a useful tool for designers to measure and improve coverage, we present several techniques for debugging what part of the projection is leading to blowup.

2 Preliminaries

Before any type of exploration can take place, it is necessary to either extract the design's control FSM or create one via some abstraction mechanism. We will assume this is done with one of the existing methods [2, 3, 6], resulting in an FSM M . M is given as 4-tuple $\langle X, I, q_0, \delta \rangle$, where X is the set of state variables $\{x_1, x_2, \dots, x_n\}$, I is the set of input variables $\{i_1, i_2, \dots, i_m\}$, q_0 is the initial state, and δ is the next state function, mapping a current state and input assignment to a next state. Σ is the set of all possible states in M .

BDDs are used to represent a set of states. Given a set of states $S(x)$, the *image* (set of states that can be reached from S in 1 step) and *preimage* (set of states that can reach S in 1 step), can be found by:

$$\begin{aligned} \text{img}(S(x)) &= \lambda x'. \exists x, y. (x' = \delta(x, y)) \cdot S(x) \\ \text{pre}(S(x)) &= \lambda x. \exists x', y. (x' = \delta(x, y)) \cdot S(x') \end{aligned}$$

The set R of reachable states in M can be found by a least fixpoint iteration $R = \text{reach}(M, q_0) = \text{lfp}_{Z. \lambda x. (q_0 \vee \text{img}(Z))}$

Let $X_I \subset X$ be the subset of state variables that are "interesting", i.e. whose exact projection we would like to find for coverage analysis and test generation. Let R_I denote the set of states reachable over the X_I variables. Formally, $R_I(x_I)$ is the projection of $R(x)$ onto X_I , i.e. $R_I(x_I) = R \downarrow x_I = \exists (X - X_I). R(x)$. Computing R exactly and then projecting onto R_I may not be feasible for large designs. However, it is feasible to compute an over-approximation of R_I . Let M_I be an FSM derived from M , with all state variables not in X_I treated as non-deterministic primary inputs. Since M_I has significantly fewer state variables than M , we can easily compute $R_I^+ = \text{reach}(M_I, q_0 \downarrow x_I)$. Clearly $R \subseteq R_I \subseteq R_I^+$

3 Projection Directed Exploration

Projection directed exploration builds the projection R_I in parallel with the exploration of R , using R_I as guide to direct which parts of R are explored. Paths in R are explored only if they lead to real states that project onto new interesting states. Because it is difficult to determine which paths in R will lead to new states in R_I when exploring forward, paths are instead searched in reverse using *backward search*.

To determine if a path exists to a new state t_I in R_I , t_I is first mapped onto the set of states $T \in \Sigma$ that project onto t_I (not all these states are necessarily in R). Successive preimages of t_I are taken until either a

preimage intersects with a state already in R , or a fixpoint is reached. If an intersection is found, a path in R can be constructed that leads to t_I by traversing forward from the intersection along the path defined by the successive preimages. If a fixpoint is reached with no intersection, then no path exists from any state in R that leads to a state projection onto t_I .

```

backward_search( $M$ , from  $from$ , to  $Target$ )
 $P_0 = Target$ ;  $Total = \emptyset$ ;
for ( $i = 1$ ;  $P_i$  not empty;  $i++$ )
     $P_i = pre(M, P_{i-1}) \wedge \overline{Total}$ 
     $Total = Total \vee P_i$ 
    if ( $P_i \wedge from$ )
        return path
    path = (path,  $P_i$ )
return failure

```

In theory, R_I can be found by enumerating all $2^{|X_I|}$ possible states in R_I and performing a backward search to check for their presence in R . In practice, the number of searches and the difficulty of each search would make this intractable for most designs. Moreover, many successful searches would duplicate work. However, a number of optimizations are possible.

3.1 Basic Improvements

The number of states searched for can be pruned significantly by using R_I^+ - an over approximation of R_I - as an initial estimate of the target states to check. The number of searches can be reduced by searching for all possible target states at once. When a target (or possibly multiple targets) is found, it is removed from the target set and the process repeated. Once a search fails, the remaining targets are unreachable.

Avoiding the work done by previous searches reduces the algorithmic complexity of the exploration. Deeper searches are also prone to BDD size blowup, as each successive preimage can pull in a larger number of state variables. Making searches incremental, by saving the real states encountered in the previous searches, gives subsequent searches a much larger set of “start” states than just q_0 to work with, generally making the depth of the searches much smaller. This avoids repetition of work, and decreases the chance that preimage sizes will blow up before a path can be found.

This suggests that the order of searches is important, so that search depths can be minimized. This can be achieved by searching for the entire set of as yet unreachable states in R_I at once. Nearby targets will be found earlier than distant ones. In fact, if all paths to a distant target must pass through a closer target, there is no point searching for both targets at once, as the closer states must be found first. This is extra work and can result in size blowup of an intermediate preimage. This can be avoided by only searching for targets that are one step away in the approximate graph R_I^+ from the states already found. Thus the target states T_I are: $T_I = img(M_I, R_I) \wedge \overline{R_I}$

3.2 Basic Algorithm

In summary, the basic projection exploration algorithm works as follows

```

 $R = q_0$ 
 $R_I = project(R, X_I)$ 
while (target ( $T_I = image(M_I, R_I) \wedge \overline{R_I}$ ) is not empty)
    path = backward_search( $M$ , from  $R$ , to  $T_I$ )
    if (path is empty) search is done
    else
         $R = R \vee path$ 
         $R_I = R_I \vee project(path, X_I)$ 

```

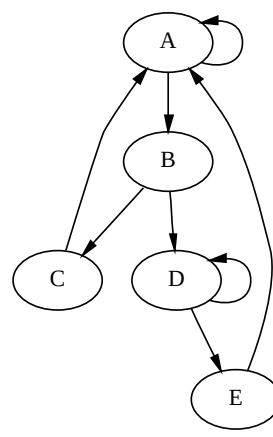


Figure 1: bottleneck diagram

4 Coping with BDD Size Blowup

We employ several techniques and heuristics to minimize BDD sizes during backward search. *Bottleneck constraints* can use the structure of R_I^+ to guarantee that a path can be found within a much smaller search space. *Reachability invariants* can prevent backward search from including clearly unreachable states. *Depth charts* and *state tags* allow the construction of tighter, per-target invariants, and allow more intelligent *depth first search* when all other techniques fail.

4.1 The Bottleneck Constraint

When searching for some states, it may be the case that we are guaranteed to find a path originating in a small portion of R . Consider fig 1: Any path originating from a real state x_D that projects onto state D must either stay in D indefinitely or enter E. When searching backwards from E, we can limit the search to just paths in D, significantly reducing search effort.

This situation arises often in designs. Consider a state that is waiting for a counter (in the controller state, but not in the interesting state) to go “off”. Depending on the size of the counter, backward search may require a large number of preimages, putting the search in jeopardy of exploring all over R and possibly blowing up. Limiting the search to D prevents this and significantly simplifies the search.

More precisely, S_B is a *bottleneck* for target states S_T if all paths originating in S_B either stay in S_B or enter S_T (i.e. $img(S_B) = S_B \vee S_T$) (It is possible that a path originating in S_B may never leave. For the control graphs of most designs, if $S_B \neq R_I$, this is probably undesirable behavior that the designer should know about.)

In the worst case, R_I itself can act as a bottleneck for a target. Intuitively, if a path being searched leaves R_I , it must at some point reenter R_I . To do this, it must pass through the target states. Since all target states are searched for in parallel, the path is already being searched and can be dropped when it leaves R_I .

4.2 Reachability Invariants

During backward search, each successive preimage may include states that are unreachable in R . This can be prevented to some extent using invariants.

An *invariant* can be found by doing reachability analysis over a subset of state variables X_{inv} from the design. (X_{inv} should be a superset of X_I to have much benefit). This is an over-approximation of R , and can be used to prune preimages during backward search. Well chosen subsets of state variables that capture critical aspects of the designs behavior can considerably reduce BDD size.

Variables for X_{inv} should be chosen to capture relationships between important state variables in the design. For example, if two interesting

FSMs communicate via another variable, placing them together in X_{inv} can capture their relationship.

When using an invariant it is important to make sure that it does not introduce variables into preimages in the process of removing unreachable states. This can happen early in the search when support sizes are small, and can lead to larger BDD sizes for preimages later in the search.

Incremental Invariants For each backward search, only a subset of the invariant is actually relevant. Because of the bottleneck state, any path found to a target T will lie in the current R_I until entering T on the last step. States that can only be reached on paths that leave and reenter R_I can be safely ignored during the backward search, and can also be removed from the invariant. We call this reduced invariant an *incremental invariant*.

For a given R_I , the incremental invariant can be found by a least fixed point of images, limiting each image to R_I . Computing this before each round of backward search is expensive. Since the R_I is a superset of the previous search's R_I , the incremental invariant is also a superset; and the previous invariant can be used in computing the new one. It is also necessary to keep track the invariant's *Frontier* – invariant states that lie outside of the current R_I .

```
compute_invariant( $R_{inv}, Frontier, R_{prev_I}, R_I$ )
   $new = R_I \wedge \overline{R_{prev_I}} \wedge Frontier$ 
   $Frontier = Frontier \wedge \overline{R_I}$ 
  while (new is not empty)
     $img = img(new)$ 
     $new = img \wedge R_I \wedge \overline{R_{inv}}$ 
     $Frontier = Frontier \vee (img \wedge \overline{R_I})$ 
     $R_{inv} = R_{inv} \vee (img \wedge R_I)$ 
  return  $R_{inv}, Frontier$ 
```

It is still more expensive to maintain the incremental invariant than to compute the complete invariant initially, but this cost is offset by simpler search.

4.3 Depth Charts

Well chosen invariants can provide useful information on the structure of the full graph R . By traversing the invariant graph, starting from the set of states that R projects onto, and keeping track of each image “layer” of the fixpoint, a *depth chart* can be built.

A lower bound on the distance of a set of states S from R can be estimated by the earliest occurrence of S in the depth chart. This estimate is useful in tracking the progress of backwards searches, and can be used in *depth first subsetting* (see below) when preimages become too large.

A constraint that contains likely paths to a target can be built by traversing backward through the depth chart, starting at points where the target intersects the chart. This “tube”, known as a *depth chart constraint* can be used during backward search to limit the state space searched. It can also be used as a guide for forward search from states already in R to the target, which can be very efficient.

4.4 State Tags

After the first preimage of a backward search is taken, the correlation between states in the preimage and the target states they correspond to is lost. By tagging each target state with a new *tag* state variable, with an identity transition function, this correspondance can be kept.

State tags allow backward search to continue after a hit by removing states in the preimage corresponding to the found target and continuing the search for the remaining targets. This avoids wasting work that has been done for the other targets: the amortized cost of computing the target, the bottleneck constraint, and the incremental invariant; along with the actual preimages taken so far.

Tags allow a depth chart constraint to be created for each target state and applied only to states in paths to that target.

Finally, the state tags can be used as an additional piece of information when making subsetting decisions during depth first search.

4.5 Depth First Search

When the above techniques fail to reduce BDDs to manageable sizes, the last resort is *depth first search*. Backward search continues on a subset of the current image, and the remainder is saved for backtracking if necessary. Several factors are considered when choosing a subset. States are grouped by their approximate distance from the target in R (based on the depth chart) and by which target state they are related to (based on the state tags). In this way, states closer to hitting can be given priority, while states corresponding to poorly behaving targets can be temporarily removed from the search so that progress can be made on the remaining targets.

When these groupings fail to sufficiently reduce BDD sizes, dense subsetting, as proposed by Ravi & Somenzi [7], is used.

5 Coping with Search Failure

Despite the above techniques, state explosion and BDD size blowup are still potential problems, especially for large designs. When these problems do occur, it is important that projection exploration still provide useful results back to the designer, so that coverage analysis and test generation can still be done. It is also important to provide feedback on the interesting states that the projection exploration was unable to find paths to, so that the designer can either rule those states out manually, or provide hints to guide the exploration in finding them.

At the end of exploration, the interesting states are in three categories: *reachable* - a real path has been found to the state, *unreachable* - the state has been proven to be unreachable, or *hard* - the state search failed due to size blowup and its reachable/unreachable status is unknown.

We have two tools to help the designer evaluate the status of the projection exploration. An FSM graph viewer can display state graphs with hard states marked allowing the designer to search for patterns and get a feel for the graph structure. To understand the paths to specific states, a symbolic path editor allows paths to be displayed and edited, providing a way to either create a path to the hard state, or understand why it is hard.

5.1 FSM Graph Display

When exploration stops, the designer is left with a set of hard states that may or may not be reachable. The first thing to do is get some intuition as to where those states are in R_I . It may be that they are spread out randomly, but in some situations they may be grouped together, unreachable for a common reason.

To help the designer visualize where the hard states lie in R_I , we have built a state graph display tool that can display graphs for individual FSMs and their cross products. When multiple FSMs are displayed at once, correlated states can be displayed. Subsets of graphs can be displayed, useful for large graphs with little structure.

Obvious patterns may appear out when looking at certain cross products of FSMs - a region of a graph that is hard might be “obviously” unreachable to the designer (based on higher order knowledge of the design). A *designer axiom* (not based on reachability analysis) can be created to keep the projection exploration from wasting effort searching for these states.

When no patterns appear, the graph display can still be useful for examining what nearby states are reachable and candidates for path analysis, described below.

5.2 Path Analysis

Given a specific hard state, it is useful to examine paths to that state in the invariant graph, or real paths to similar states. To support this we have built a symbolic path editor. It displays paths and allows them to be edited. After changing input values, state values, the trace is updated using symbolic techniques. Intermediate nodes in the design (wires that are a combinational function of the current state) can also be displayed.

If a hard state is similar to another state that was reached, the designer may either investigate the real path to reached state to determine if that

path can be modified to reach the hard state, or learn why the hard state is not so similar after all. This examination is often a useful exercise for the designer in understanding their own design.

Another possibility is to build a path to the hard state in the invariant graph. State variables outside the invariant support will appear as pseudo-inputs. The designer can then examine the values required on those pseudo-inputs, checking for inconsistencies that might be captured with a new specific invariant. They can also push a real state forward through the path and examine at which steps the state cannot go forward.

6 Results

These techniques were implemented in the *vex* toolbox [1] (which relies on the CUDD BDD package [8]) and evaluated on the FLASH (FLexible Architecture for SHared Memory) Protocol Processor (PP) [5]. The PP is an embedded, dual-issue, 64 bit, MIPS core used to execute shared memory protocol code. It is a typical RISC processor with a 5-stage pipeline, 1 MB offchip data cache, and 16KB on chip instruction cache. It interfaces with other units on the FLASH node controller chip via a memory interface and a general purpose bus.

The PP control logic was extracted and appropriate environment models created for external units (primarily on chip memory controller handles DRAM access). The total controller model has 146 state variables. A subset of 11 state variables – comprising several state machines that handle load/store/cache instructions, process cache misses and direct the load/store datapath – was identified as the *interesting* state. The real graph is too big to explore, the approximate graph for the interesting variables has 432 states.

Because this design was the motivating example for the techniques in this paper, the exploration of the interesting projection was a continuous process of exploring, analyzing hard to reach parts of the projection, improving invariants, and improving the techniques themselves. Beside improvement of techniques, the same process will be necessary when handling any large design.

Before exploration, an initial designer invariant was built. Starting with the interesting FSMs, variables that captured communication between the FSMs or captured the relationship between common inputs to the FSMs (such as 1-hot instruction signals) were added. This invariant was improved to the point that its projection onto the interesting variables looked reasonable to the designer. As projection exploration was done, the paths to hard to reach states were examined by the designer and new variables added that would increase the precision of the invariant to rule out those states all together. Eventually, the support of the designer invariant grew to 66 variables, still reasonable to compute, and ruled out 145 states from the approximate graph.

Even with this improved invariant, exploration was only finding 220 states out of the 287 predicted by the invariant. By projecting the 70 hard states onto smaller cross products of the four interesting FSMs (i.e. 2 and 3 machine crossproducts), the designer was able to make some useful observations.

First, one group of hard states involved an FSM that counts pending stores that are queued and waiting to be written to the dcache while another machine was in a state where those stores should have been written out. Although the projection exploration could not prove these states unreachable (the searches for them were failing due to blowup), the fact that they were too hard to reach was consistent with the designer's belief that they should be unreachable.

A similar situation occurred for a set of hard states when viewing the crossproduct between the machine responsible for spilling queued stores when a load to conflicting address occurs and the machine counting the queued stores. Several states, where all stores should have been flushed, still had a store pending. The designer thought it was reasonable that these states were unreachable.

For both these cases, specific axioms were added to mark these states unreachable. Since these hard states would almost certainly represent error states if they could be reached, these axioms can be thought of as proof obligations that need to eventually be discharged (if possible).

Table 1: Number of interesting states

source	# states
Approx graph	432
approx + invariant	287
approx + invariant + axioms	227
Actually reached by projection exploration	227
Reached in test vectors	72 (31.7%)

The remaining hard states fit no particular pattern, although the examination of them was a useful exercise. In one case, a hard state was similar enough to a reached state that the designer thought that a real path to the reached state could be modified to go to the hard state. In examining the path, the designer noticed that if the system stalled at the right time, two different FSMs that should make a decision based on a variable at the same time would actually make their decisions at different times. Since the variable is not preserved during the stall, this is probably a bug in the design. We are currently looking into this further.

With the specific invariants removing some hard states from consideration and a loosening of some backtracking limitations on the search, the remaining 10 states were found, pushing the “coverage” of the projection exploration upto 227 states out of the 227 that seemed reasonable to the designer. Table 1 summarizes this process. The edge coverage is currently 798 edges with 145 hard edges that we are in the processing of analyzing.

7 Conclusions

Previous work has demonstrated that accurately exploring a projection from a much larger graph is a useful technique for coverage analysis and test generation, since it allows the designer to deal with the complete hardware description, yet focus his/her attention on a specific collection of state machines. The problem with this technique is that for many interesting situations the complete state graph is too large to completely generate. This paper has explored a number of techniques to allow one to use the approximate projection to direct the state exploration, exploring only that portion of the state space where the designer interest is currently focused. These techniques have allowed us to explore the cache control FSM of an embedded processor, a system that is too complex to analyze by other methods.

Even with the projection directed search, dealing with the complete state space can lead to BDD size blowups. To deal with these issues, we have created a number of interactive tools to try to provide information back to the designer, both to provide that partial information that has been obtained so far, and to enable them to provide additional constraints that may help the search. While this technique is promising, currently it requires a very educated user – one who understands both the design and the working of the tool. Further work is needed to enable this type of tool to be useful for a standard validation engineer.

References

- [1] J.P. Bergmann and M.A. Horowitz. “Vex - A CAD Toolbox”. In *DAC*, June 1999.
- [2] D. Geist, M. Farkas, A. Landver, and Y. Lichtenstein. “Coverage-Directed Test Generation Using Symbolic Techniques”. In *FMCAD*, November 1996.
- [3] R. Ho and M. Horowitz. “Validation Coverage Analysis for Complex Digital Designs”. In *ICCAD*, November 1996.
- [4] R.C. Ho, C.H. Yang, M.A. Horowitz, and D.L. Dill. “Architectural Validation for Processors”. In *ISCA*, June 1995.
- [5] J. Kuskin and et al. “The Stanford FLASH Multiprocessor”. In *ISCA*, June 1994.
- [6] D. Moundanos, J.A. Abraham, and Y.V. Hoskote. “Abstraction Techniques for Validation Coverage Analysis and Test Generation”. In *IEEE Transactions on Computers*, January 1998.
- [7] K. Ravi and F. Somenzi. “High-Density Reachability Analysis”. In *ICCAD*, November 1995.
- [8] F. Somenzi. *CUDD: CU Decision Diagram Package*. <http://vlsi.colorado.edu/pub/>.