Improved Approximate Reachability using Auxiliary State Variables *

Shankar G. Govindaraju, David L. Dill and Jules P. Bergmann
Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{shankar@encore, dill@cs, bergmann@cs}.stanford.edu

Abstract

Approximate reachability techniques trade off accuracy for the capacity to deal with bigger designs. Cho *et al* [4] proposed partitioning the set of state bits into mutually disjoint subsets and doing symbolic forward reachability on the individual subsets to obtain an overapproximation of the reachable state set. Recently [7] this was improved upon by dividing the set of state bits into various subsets that could possibly overlap, and doing symbolic reachability over the overlapping subsets. In this paper, we further improve on this scheme by augmenting the set of state variables with auxiliary state variables. These auxiliary state variables are added to capture some important internal conditions in the combinational logic. Approximate symbolic forward reachability on overlapping subsets of this augmented set of state variables yields much tighter approximations than earlier methods.

1 Introduction

Binary Decision Diagrams (BDDs) [2] have enabled formal verification to tackle larger hardware designs than before. Using BDDs to represent sets of states has enabled symbolic forward reachability techniques to enumerate the state space of bigger designs. However for many large design examples, even the most sophisticated BDD-based verification methods cannot produce exact results because of BDD-size blowup. Hence, we settle for approximate reachability.

An overapproximation (*i.e* superset) of the reachable states can still be very useful. If an assertion holds for the approximate reachable states, it is guaranteed to hold in the exact reachable set. It can also be used to simplify symbolic model checking efforts, by preventing [8] the model checking algorithms from exploring unreachable states. Further, the approximate reachable set provides *don't cares*, that can be used in synthesis.

*This work was supported by DARPA contracts DABT63-94-C-0054 and DABT63-96-C-0097. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

DAC 99, New Orleans, Louisiana

1.1 Comparison with Related Work

Various approaches to approximate reachability and verification using BDDs have preceded this work. Cho et al [4, 5] proposed approximate algorithms to do symbolic forward reachability. Their basic idea was to partition the set of state bits into *mutually disjoint* subsets, and then do a symbolic forward propagation on each individual subset. This was further generalized [7] by allowing for *overlapping projections*. In this scheme, the set of state bits was divided into various subsets that could overlap.

This paper further generalizes and improves on existing approximate symbolic reachability schemes, by augmenting the set of state variables with some auxiliary state variables. An auxiliary variable is an internal state component that is added to the implementation without affecting the externally visible behavior. These extra state variables typically represent important internal abstractions used by designers.

The idea of augmenting a legal implementation with some extra state components in a way that places no constraints on the behavior of the implementation is not entirely new. Abadi and Lamport [1] introduced a special class of auxiliary variables, *history* and *prophecy* variables, to broaden the applicability of refinement mapping techniques. We propose using auxiliary state variables to broaden applicability of approximate reachability techniques.

Consider the simple design shown in figure 1. The design has 96 state variables, denoted by $(x_1, \ldots, x_{96})$. The *Equality Detector* checks whether the two input bit vectors are identical and passes its output to the control state machine. Exact reachability would require computing images over the variables $(x_1, \ldots, x_{96})$. Intermediate image BDDs with such large support sets often blow up. Alternatively we could choose to do approximate reachability over the disjoint [4] subsets $(x_1, \ldots, x_{32})$, $(x_{33}, \ldots, x_{64})$ and $(x_{64}, \ldots, x_{96})$. Since the subsets have 32 variables, the intermediate image BDDs have 32 variables in their support and are less likely to blow up, but there is a price in loss of accuracy since interaction between the variables in different subsets is lost.

Using overlapping projections [7], we could capture some interaction by choosing the subsets $(x_1, \ldots, x_{64})$,

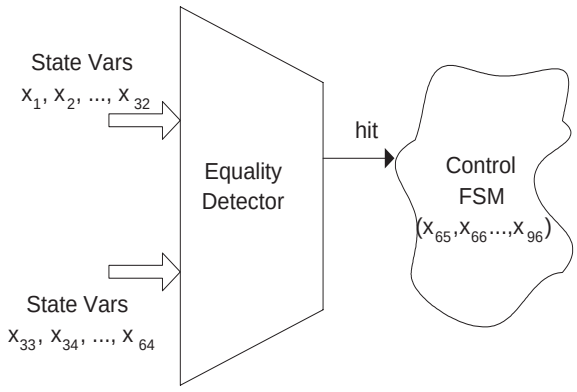

Figure 1: Example to illustrate potential of using auxiliary variables

$(x_{33}, \ldots, x_{96})$ and $(x_1, \ldots, x_{32}, x_{65}, \ldots, x_{96})$. The intermediate image BDDs have 64 variables in their support, but it captures more interaction between the state variables than the disjoint partition case.

However, the only interaction between state variables $(x_{65}, \ldots, x_{96})$ and the other state variables happens through the signal *hit*. By introducing an auxiliary state variable for the wire *hit*, interaction between the state variables is captured by choosing the subsets $(x_1, \ldots, x_{32})$, $(x_{33}, \ldots, x_{64})$, $(x_{65}, \ldots, x_{96}, hit)$, and doing symbolic reachability [7] over them. The largest subset in this case is of size 33, but it captures the critical correlation between all 96 state variables in the design.

The contribution of this paper is to show how *auxiliary* state variables enable more refined approximate reachability. The new scheme is more general than *overlapping projections* [7] which in turn is more general than *disjoint partitions* [4]. Significant improvement is obtained when this simple enhancement is applied to several control modules from the I/O unit in the Stanford FLASH Multiprocessor, and to the larger ISCAS89 circuits.

2 Background

We analyze synchronous hardware, given as a Mealy machine $M = \langle x, y, q_0, \mathbf{n} \rangle$, where $x = \{x_1, \ldots, x_k\}$ is the set of state variables, and y is the set of input signals. The set of states is given by $[x \to \mathcal{B}]$, where $\mathcal{B} = \{0,1\}$. The initial state q_0 is in $[x \to \mathcal{B}]$. The next state function is $\mathbf{n} : [x \to \mathcal{B}] \times [y \to \mathcal{B}] \to [x \to \mathcal{B}]$. BDDs can be used to represent sets and manipulate them [3]. Let $R(x)$ (a BDD with support in x) represent a set of states, then image of R under $\mathbf{n}$ is computed as,

$$Im(R(x), \mathbf{n}(x,y)) = \exists x, y.(x' = \mathbf{n}(x,y)) \wedge R(x).$$

Let $\mathbf{w} = (w_1, \ldots, w_p)$ be a collection of not necessarily disjoint subsets of x. We define the operator $\alpha_j(R)$ which projects a BDD $R(x)$ onto the variables in w_j. Let z consist of all of the Boolean variables in x that are *not* in w_j. We can define α_j as

$$\alpha_j(R(z, w_j)) = \exists z.R(z, w_j).$$

The projection operator α projects a BDD $R(x)$ onto the various w_j's, and the concretization operator γ conjoins the collection of projections.

$$\begin{aligned} \alpha(R(x)) &= (\alpha_1(R), \ldots, \alpha_p(R)). \\ \gamma(R_1, \ldots, R_p) &= \bigwedge_{j=1}^{p} R_j. \end{aligned}$$

The operator α allows us to represent a big BDD with support in x by a list of potentially smaller BDDs with limited support, at the cost of loss of accuracy. The operator γ can potentially result in a bigger BDD with bigger support, hence we would like to avoid computing $\gamma(R_1, \ldots, R_p)$ explicitly. During approximate reachability, the intermediate images are stored as an *implicit* conjunction of the elements of a list of BDDs, $\mathbf{R} : (R_1, \ldots, R_p)$, where each R_j has support in w_j.

The key operation is the approximate image computation: Given an *implicit* conjunction of BDDs $\mathbf{R} : (R_1, \ldots, R_p)$, compute a list $\mathbf{S} : (S_1, \ldots, S_p)$ whose implicit conjunction is the set of states than can be approximately reached in one step using the next state functions $\mathbf{n}$. More formally $\mathbf{S} = \alpha(Im(\gamma(\mathbf{R}), \mathbf{n}(x,y)))$. An efficient algorithm to compute $\mathbf{S}$ was proposed in DAC98 [7], which we also use here.

Starting from the initial state q_0, then repeatedly computing approximate images until we reach a fixed point gives an overapproximation of the reachable state set. A more formal treatment was given in DAC98 [7]. (Even though we have a single initial state, the method can be applied to any arbitrary choice of initial states).

3 Auxiliary State Variables

An auxiliary state variable is useful because it captures important properties of many state variables into a *single* new state bit. This can be added to the other subsets to capture correlation between many state variables, even as the number of variables in different subsets is small.

3.1 What can be an Auxiliary State Variable?

We make use of auxiliary variables by converting them to state variables. A next state function is assigned to each of them as in the following example.

A typical hardware design, as shown in figure 2, has a set of state holding elements ((x_1, x_2, x_3) in figure 2) and some combinational logic. Each state variable has an associated next state function logic ((n_1, n_2, n_3) in figure 2). Let a be some internal wire in the design, and let $a = g(x)$ be the function that determines the

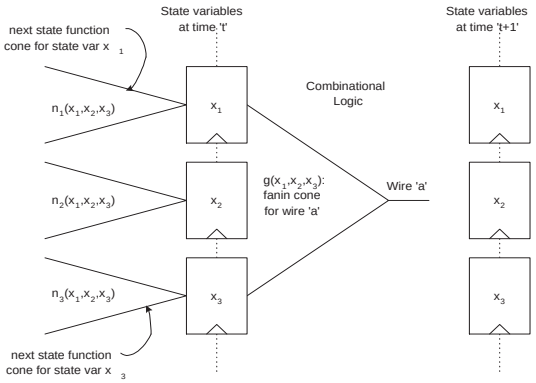

Figure 2: Typical Design

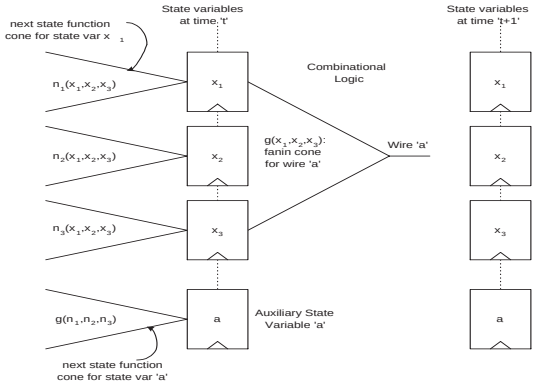

Figure 3: Design including Auxiliary State Variables

value of a in time t as a function of the state variables x at time t.

If we let the subscript denote the time stamp, we have: $a_t = g(x_t)$ and $a_{t+1} = g(x_{t+1})$. Using $x_{t+1} = \mathbf{n}(x_t, y_t)$, we get $a_{t+1} = g(\mathbf{n}(x_t, y_t))$, which is the required next state function for auxiliary state variable a. This transformation is shown in figure 3. Note that we would not have been able to do the transformation above if g involved some input variables in its support. If $a = g(x, y)$ (where y is the input bits) then $a_{t+1} = g(x_{t+1}, y_{t+1})$ and we cannot represent the inputs in the next cycle, y_{t+1}, in terms of x_t and y_t.

We conjecture this limitation can be circumvented by including the inputs as part of the state (as in a Kripke structure). We never used this for any of our results here, but the Mealy machine $M = \langle x, y, q_0, \mathbf{n} \rangle$, can be transformed to another Mealy machine $M' = \langle x', y', q_0', \mathbf{n}' \rangle$, where $x' = x \cup y$ and the initial condition $q_0' = q_0$. The y' component is a set with a primed version for each variable in y. The next state function for the x state variables remains the same, but for the y variables, their next state function is the corresponding input variable from y'. Assuming totally unconstrained input environment, the machines M and M' allow the same externally visible behaviors and hence have the same set of reachable states (projected on to the x variables). However M' allows us more flexibility in choosing auxiliary state variables.

3.2 Initial Condition for Auxiliary State Variables

The auxiliary state variables need to be initialized. Let $\mathbf{a} : (a_1, \ldots, a_m)$ be the list of auxiliary variables and $\mathbf{g} : (g_1, \ldots, g_m)$ be the list of Boolean functions (represented as BDDs) such that $g_i(x)$ determines the value of a_i at time t in terms of state variables x at time t. The initial condition for the $\mathbf{a} : (a_1, \ldots, a_m)$ variables is obtained by the following image computation, $Im(q_0, \mathbf{g})$. In our applications, initial condition q_0 is a single state, and this reduces to computing $g_i(x) \downarrow q_0$ for each auxiliary variable a_i. (The $\downarrow$ is the generalized cofactor [6] operator).

3.3 Heuristics to Choose Auxiliary State Variables

Our scheme for choosing which internal abstractions to convert to auxiliary state variables is presently manual, and relies on being able to inspect the RTL source. We believe that it helps to look at the RTL source, because designers often create internal abstractions themselves, while coding up their design using a hardware description language (such as Verilog). Hence we can take leverage off this high level information directly by inspecting the RTL description.

First, we find the FSMs by inspecting the Verilog source. The next state transition for every FSM was typically encoded as part of an *always* block in the Verilog source. By inspecting the *always* block it is possible to extract the internal wires that affect the next state transition of each FSM, and if those internal wires in turn depend on many state variables they are chosen as auxiliary state variables.

However the gate level descriptions of circuits like the ISCAS 89 benchmark circuits are devoid of any high level information. For such circuits, we look for internal wires which have a *high fanin* and *high fanout*, and are at the same time solely determined by the state variables in the design (*i.e* their fanin cones involve only state variables). The intuition behind our heuristic is that such high fanin internal wires carry some information about the large number of state variables in their fanin cone. Hence including these wires as auxiliary state variables in other subsets of $\mathbf{w}$, allows us to capture some correlation between the state variables in the other subsets and the large number of state variables in the fanin cone of the internal wire.

4 Experiments

The method was evaluated on a collection of control circuits from the MAGIC chip, a custom node controller ASIC in the Stanford FLASH Multiprocessor [9]. The circuits are control intensive, and the state bits do not include data path bits. Table 1 gives a brief description of the sizes of various control modules ex-

tracted from the I/O unit, in terms of the number of state variables, auxiliary state variables and input variables. (IOQ_ReqD stands for the module obtained by combining the submodules IOInboxQCtl and ReqDecode, whereas ReqS_ReqD stands for the module obtained by combining ReqService and ReqDecode). (The results for these modules appear in the same order in Table 2). We were unable to find the exact reachable set for any of these control modules.

Table 1: Control Modules in I/O unit in FLASH

Module	State	Auxiliary	Total	Input
IOQ_ReqD	60	6	66	25
ReqS_Req	78	14	92	48
PciInterface	88	20	108	55

The experimental implementation of the method was in LISP, calling David Long's BDD package (implemented in C) via the foreign function interface. Our approximate algorithm returns a superset of the reachable states. To quantify the size of the superset, we compute the satisfying fraction of the the superset. (Please refer to the appendix for the algorithm that was used to compute an upper bound on the satisfying fraction). Since projection induces an over-approximation, smaller satisfying fraction indicates better results.

We compare our results with the earlier reported numbers obtained with overlapping projections of the usual state variables alone. The same variable ordering was used for both the schemes. The maximum number of nodes for each experiment is preset at *Node Limit* and we try to get the best results using the two schemes (overlapping projections of usual state variables alone *vs* overlapping projections of augmented set of state variables). *Node Count* keeps track of the largest number of nodes that existed at a time during the experiment. The *Time* column lists the cpu time (in seconds) needed to reach the fixed point on a MIPS R4300 with 768MB of RAM. *Sat_fr* records the size of the approximate reachable state set (a superset) in terms of satisfying fraction. The last column under the heading *Relative* is the ratio between the satisfying fraction obtained by using usual state variables alone and the satisfying fraction obtained on adding auxiliary variables. Thus, larger figures in the *Relative* column indicate better results with auxiliary variables.

The results in Table 2 show that the use of overlapping subsets over the augmented set of state variables is very effective at improving over-approximations of the reachable state set. The improvement is at the expense of some increase in the BDD node count. However, it would not be possible to obtain such a tight approximation using overlapping subsets over the usual state variables alone, since that would require prohibitively large subsets, resulting in BDD blowup problems.

4.1 ISCAS Benchmarks

We have also tried our algorithm on the larger circuits from ISCAS 89 benchmark suite. We use the partitions used by Cho *et al* [4] to identify the FSMs in the design. To these partitions, small overlaps were added to report the numbers in DAC98 [7] to show the potential of approximate reachability on overlapping subsets of the usual state variables. Here, we further add some auxiliary state variables to some of the overlapping subsets, and compare with the recently reported results in [7]. Table 3 gives a brief description of the size of the various benchmark circuits used in this work. (We omit s1238 because it is a small circuit amenable to exact traversal. We are unable to report comparative figures for s35932 because we could not procure the partitions used by Cho et al for s35932). We tried our new algorithm on s1423, but unfortunately could not improve on the results reported in [7]. (We suspect it is because s1423 has a highly interconnected STG. Some high level insight into the design, which ISCAS benchmark circuits are devoid of, could better guide the choice of auxiliary variables). However for s13207, s15850 and s38584, we report improvement by at least an order of magnitude.

Table 3: Large Circuits from ISCAS 89 Suite

Circuit	State	Auxiliary	Total	Input
s13207	669	39	708	31
s15850	597	14	611	14
s38584	1452	12	1464	12

Given the large number of state variables in these circuits, and that we allow for overlaps among the various subsets, it is very difficult to compute the size of the approximate reachable set. The numbers in Table 4 under the *Sat Fr* column for *Auxiliary Variables* are *upper bounds* on the size of the reachable set. (Please refer to the appendix for the algorithm used to compute an upper bound on the size of the approximate reachable set). We believe that the true size of the approximate reachable set using auxiliary state variables, is much smaller than what we report here.

Note that we use TMBM algorithm [4] for these benchmarks. TMBM starts off as TFBF [4] and then switches to MBM [4] after a few iterations. The *Iter* column in Table 4 lists the number of iterations of doing TFBF + the number of iterations in the outer greatest fixpoint of MBM.

5 Conclusions

Our experiments show that a few appropriately chosen internal conditions added as auxiliary variables can substantially improve the quality of the overapproximation. We need to look at automatic methods to choose collection of subsets for gate level descriptions.

Table 2: FLASH I/O Circuits: Size of Approximate Reachable Set

Node Limit	Usual State Variables			Adding Auxiliary Variables			Relative
	Sat. Fr.	Time	Node Count	Sat. Fr.	Time	Node Count	
100,000	2.570e-08	22.65	63,180	1.485e-09	47.62	97,685	1.731e+01
150,000	"	"	"	1.399e-09	60.52	111,517	1.838e+01
Node Limit	Usual State Variables			Adding Auxiliary Variables			Relative
	Sat. Fr.	Time	Node Count	Sat. Fr.	Time	Node Count	
1,000,000	3.835e-09	553.34	644,667	2.632e-09	1,301.88	846,476	1.457
2,000,000	"	"	"	2.282e-09	1,232.27	1,832,354	1.680
Node Limit	Usual State Variables			Adding Auxiliary Variables			Relative
	Sat. Fr.	Time	Node Count	Sat. Fr.	Time	Node Count	
1,000,000	1.801e-05	308.18	466441	5.892e-06	1,471.88	971,880	3.057
10,000,000	2.175e-06	2,907.86	1,260,260	7.003e-07	9,174.01	8,349,050	3.105

Table 4: ISCAS 89 Circuits: Size of Approximate Reachable Set

Circuit	Usual State Variables			Adding Auxiliary Variables			Relative
	Sat. Fr.	Iter	Node Count	Sat. Fr.	Iter	Node Count	
s13207	1.136e-115	10+5	198779	1.241e-117	10+5	1171473	9.155e+01
s15850	3.938e-102	10+4	336048	3.918e-103	10+4	339031	1.005e+01
s38584	5.764e-57	10+5	1853461	1.198e-58	10+4	1952730	4.813e+01

6 Appendix

6.1 *Sat_Fr* of Superset for FLASH I/O circuits

Given $\mathbf{S} : (S_1, \ldots, S_p)$, corresponding to the collection of possibly overlapping subsets $\mathbf{w} : (w_1, \ldots, w_p)$, we want to compute *sat_fr* of $\gamma(\mathbf{S})$. Let $a : (a_1, \ldots, a_m)$ be the set of auxiliary state variables. Corresponding to each auxiliary state variable a_i, let $g_i(x)$ be the Boolean function (represented as a BDD) which determines the value of the auxiliary state variable a_i in time t as a function of the value of the usual state variables at time t. Our algorithm substitutes the function g_i for every instance of a_i in the elements of the list $\mathbf{S}$. At this point $\mathbf{S}$ has only the usual state variables in its support. The algorithm then explicitly computes $\gamma(\mathbf{S})$ and finds its satisfying fraction.

for *j=1* **up to** *p* **by** *1* **do**
  for *i=1* **up to** *m* **by** *1* **do**
    Substitute g_i for every instance of a_i in S_j
  endfor
endfor
Compute *final_bdd* = $\wedge_{j=1}^{p} S_j$
return *sat_fr (final_bdd)*

For the larger ISCAS 89 benchmark circuits it is not feasible to explicitly compute $final\_bdd = \gamma(\mathbf{S})$. Hence we use the conservative algorithm given in [7] and we normalize the result, to compensate for increase in number of state variables. (If m is the number of auxiliary state variables added, we multiply the result obtained from the algorithm in [7] with 2^m to obtain an upper bound on the satisfying fraction for the reachable states over the usual state variables alone). An alternative method, Monte Carlo simulation technique appears to be ineffective because of the extreme sparsity of the state space covered by $\gamma(\mathbf{S})$.

References

[1] Abadi, M. and Lamport, L., "The Existence of Refinement Mappings," *LICS*, pp. 165-177, July 1988.

[2] Bryant, R. E., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677-691, August 1986.

[3] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D, L, and Hwang, L. J., "Symbolic Model Checking: 10^{20} States and Beyond," *LICS*, pp. 428-439, 1990.

[4] Cho, H., Hachtel, G., Macii, E., Pleisser, B., and Somenzi, F., "Algorithms for Approximate FSM Traversal Based on State Space Decomposition," *IEEE TCAD*, Vol. 15, No. 12, pp. 1465-1478, December 1996.

[5] Cho, H., Hachtel, G., Macii, E., Poncino, M., and Somenzi, F., "Automatic State Space Decomposition for Approximate FSM Traversal Based on Circuit Analysis," *IEEE TCAD*, Vol. 15, No. 12, pp. 1451-1464, December 1996.

[6] Coudert, O., and Madre, J. C., "A Unified Framework for the Formal Verification of Sequential Circuits," *ICCAD*, pp. 126-129, 1990.

[7] Govindaraju, G. S., Dill, D. L., Hu, A. J, and Horowitz, M. A., "Approximate Reachability with BDDs Using Overlapping Projections," *DAC*, pp. 451-456, 1998.

[8] Govindaraju, G. S. and Dill, D. L., "Verification by Approximate Forward and Backward Reachability," *ICCAD*, pp. 366-370, 1998.

[9] Kuskin, J. et al, "The Stanford FLASH Multiprocessor," *ISCA*, pp. 301-313, April 1994.