Approximate Symbolic Model Checking using Overlapping Projections

Shankar G. Govindaraju

Computer Systems Laboratory
Stanford University, Stanford, CA USA

David L. Dill

Computer Systems Laboratory
Stanford University, Stanford, CA USA

Abstract
Symbolic Model Checking extends the scope of verification algorithms that can be handled automatically, by using symbolic representations rather than explicitly searching the entire state space of the model. However even the most sophisticated symbolic methods cannot be directly applied to many of today's large designs because of the state explosion problem. Approximate symbolic model checking is an attempt to trade off accuracy with the capacity to deal with bigger designs. This paper explores the idea of using overlapping projections as the underlying approximation scheme. The idea is evaluated by applying it to several modules from the I/O unit in the Stanford FLASH Multiprocessor, and some larger circuits in ISCAS89 benchmark suite.

1 Introduction

The ability to enumerate the set of states reachable from a certain state, and the ability to enumerate the set of states that can reach a certain state are essential to many model checking algorithms. Binary Decision Diagrams (BDDs) [2] have proved to be a viable data structure for doing symbolic reachability on larger hardware designs than before. However for many large design examples, even the most sophisticated BDD-based verification methods cannot produce exact results because of size blowup. However, required properties of a design rarely rely on every implementation detail of the design, so approximate verification algorithms may yield meaningful results while handling larger designs.

1 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.
We are interested in safety properties that hold for every member of a set $S$ of states. A superset $S_{ap}$ of $S$ is called an overapproximation of $S$. Although $S_{ap}$ may be larger than $S$, it may also have a smaller representation, so the computation of $S_{ap}$ may be more efficient than $S$. If every state in $S_{ap}$ satisfies a property, we can be sure that every state in $S$ also satisfies the property. Hence, a sufficiently accurate approximation can yield a useful result.

The approximation used is based on overlapping projections of sets of states. A set of states is represented by a list of BDDs, each element of the list constrains possibly overlapping subsets of the state variables. The projection of a set $S$ of bit vectors onto a set of one-bit variables, $w_j$, is the (larger) set of bit vectors that match some member of $S$ for all variables in $w_j$ (the values of other variables are ignored). $S$ can be approximated by projecting it onto many different subsets of the variables, and considering $S_{ap}$ to be the intersection of all of the approximations.

The idea is evaluated on several control modules from a real, large design unit in the Stanford FLASH Multiprocessor, with promising results. Properties in the design were either shown to hold for all reachable states, or actual violations were proved to exist in the exact reachable state space (some violated assertions resulted from omitting constraints on the possible inputs to the design).

2 Related Work

At a high level, this work is quite similar to that of Wong-Toi, et al. [8], who used successive forward and backwards overapproximations and underapproximations to verify real-time systems. That work used polyhedra for representing sets of real numbers along with BDDs, but approximation was used only for the polyhedra, not for the BDDs.

Various approaches to approximate reachability and verification using BDDs have preceded this work. Ravi et al [16] use "high density" BDDs to compute an underapproximation of the forward reachable set. Cho et al [5] proposed symbolic forward reachability algorithms that induce an overapproximation. They partition the set of state bits into mutually disjoint subsets, and do a symbolic forward propagation on individual subsets. Cabodi et al [4] combine approximate forward reachability with exact backward reachability. Lee et al [14] propose "tearing" schemes to do approximate symbolic backward reachability. They also partition the set of state bits into mutually disjoint subsets. They form the block sub-relations for the various subsets, and then incrementally "stitch" the block sub-relations together until the approximated next state relation is accurate enough to prove or disprove a given property. In contrast to the approaches in [16] we are interested in computing overapproximations (supersets). In contrast to the approaches in [4,5,14], we allow for overlapping subsets, as overlapping projections have been shown [10] to be a more refined approximation compared to earlier schemes based on disjoint partitions.
3 Background

We analyze synchronous hardware, given as a Mealy machine \( M = (x, y, q_0, n) \), where \( x = \{x_1, \ldots, x_k\} \) is the set of state variables, and \( y \) is the set of input signals. We will use \( x' = \{x'_1, \ldots, x'_k\} \) to denote the next state versions of the corresponding variables in \( x = \{x_1, \ldots, x_k\} \). The set of states is given by \([x \rightarrow \mathcal{B}]\), where \( \mathcal{B} = \{0, 1\} \). The initial state \( q_0 \in [x \rightarrow \mathcal{B}] \). The next state function is \( n : [x \rightarrow \mathcal{B}] \times [y \rightarrow \mathcal{B}] \rightarrow [x \rightarrow \mathcal{B}] \).

In our applications, sets can be viewed as predicates, since we can form the characteristic function corresponding to a set. BDDs can be used to represent predicates and manipulate them [3]. For example, let \( R(x) \) be a predicate with support in \( x \), we can compute the image of \( R \) under \( n \) as

\[
\text{Im}(R(x), n(x, y)) = \lambda x'. \exists x, y. (x' = n(x, y)) \land R(x).
\]

Let \( g \) be a user specified property, and \( \bar{g} \) denote the complement of \( g \). Then the preimage of \( \bar{g}(x) \), i.e., the set of states that can reach a state violating the property \( g \) in one step, can be computed as follows:

\[
\text{Pre}(\bar{g}, n) = \lambda x. \exists x', y. (x' = n(x, y)) \land \bar{g}(x').
\]

3.1 Approximation by Projections

Let \( w = (w_1, \ldots, w_p) \) be a collection of not necessarily disjoint subsets of \( x \). (Each subset will be referred to as a block). We define the operator \( \alpha_j(R) \) which projects a predicate \( R(x) \) 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 \( \alpha_j \) as

\[
\alpha_j(R(z, w_j)) = \lambda w_j. \exists z. R(z, w_j).
\]

Clearly the set of Boolean vectors satisfying \( R \) is a subset of those satisfying \( \alpha_j(R) \). This can be written using logical implication as \( R \rightarrow \alpha_j(R) \). The projection operator \( \alpha \) projects a predicate \( R(x) \) onto the various \( w_j \)'s, and its associated concretization operator \( \gamma \) joins the collection of projections.

\[
\alpha(R(x)) = (\alpha_1(R), \ldots, \alpha_p(R)).
\]

\[
\gamma(R_1, \ldots, R_p) = \bigwedge_{j=1}^p R_j.
\]

Lemma 3.1 For every predicate \( R(x) \) and collection of subsets \((w_1, \ldots, w_p)\) of \( x \), \( R \rightarrow \gamma(\alpha(R)) \).

The proof for this lemma is simple since \( R \rightarrow \alpha_j(R) \) for all \( j \). Thus projecting a predicate \( R \) onto a collection of subsets, and then concretizing the projections by \( \gamma \) results in an overapproximation.

It is interesting to note that the pair of functions \((\alpha, \gamma)\) form a Galois connection [7] between the partially ordered set describing the concrete space \([x \rightarrow \mathcal{B}], \subseteq\) and the poset describing the abstract space \( \mathcal{P}([w_1 \rightarrow \mathcal{B}]) \times \ldots \times \mathcal{P}([w_p \rightarrow \mathcal{B}]), \subseteq\) where \( \mathcal{P}(S) \) denotes the power set of \( S \), and the ordering relation for the abstract space is defined as \( (R_1, \ldots, R_p) \subseteq (S_1, \ldots, S_p) \) iff \( \forall i \in [1 \ldots p], R_i \subseteq S_i \).
Let \( R = (R_1, \ldots, R_p) \) and \( S = (S_1, \ldots, S_p) \) be two tuples of equal size. We define the meet (\( \cap \)) and join (\( \cup \)) operator between \( R \) and \( S \) as follows:

\[
(R_1, \ldots, R_p) \cap (S_1, \ldots, S_p) = (R_1 \land S_1, \ldots, R_p \land S_p)
\]

\[
(R_1, \ldots, R_p) \cup (S_1, \ldots, S_p) = (R_1 \lor S_1, \ldots, R_p \lor S_p)
\]

Given the ordering relation (\( \leq \)) in the abstract domain, it is easy to verify that the join operator returns the least upper bound, and meet returns the greatest lower bound of the two elements \( R \) and \( S \) in the abstract domain. Further \( \gamma(R) \cup \gamma(S) \leq \gamma(R \cup S) \), which makes the join operator an approximation of set union. (However, the meet operator is an exact set intersection operator, since \( \gamma(R) \cap \gamma(S) = \gamma(R \cap S) \).

The operator \( \alpha \) allows us to represent a big BDD with support in \( x \) by a tuple of potentially smaller BDDs with limited support, at the cost of loss of accuracy. \( \gamma \) can potentially result in a bigger BDD with bigger support, hence we would like to avoid computing \( \gamma(R_1, \ldots, R_p) \) explicitly. Let \( \text{Im}_{ap} \) (the subscript \( ap \) denotes "approximate") return the projected version of the image of an implicit conjunction of BDDs, and let \( \text{Pre}_{ap} \) return the projected version of the preimage of an implicit conjunction of BDDs.

\[
\text{Im}_{ap}(R, n) = \alpha(\text{Im}(\gamma(R), n(x, y)))
\]

\[
\text{Pre}_{ap}(R, n) = \alpha(\text{Pre}(\gamma(R), n(x, y)))
\]

Using \( \text{Im}_{ap} \), we can compute an overapproximation, \( \text{FwdReach}_{ap}(q_0) \), of the reachable states for a machine \( M \). Analogously using \( \text{Pre}_{ap} \), we can compute an overapproximation, \( \text{BackReach}_{ap}(\bar{g}) \), of the set of states in \( M \) that can reach the set of states \( \bar{g} \) as follows:

\[
\text{FwdReach}_{ap}(q_0) = \text{Ifp } R.(\alpha(q_0) \cup \text{Im}_{ap}(R, n))
\]

\[
\text{BackReach}_{ap}(\bar{g}) = \text{Ifp } R.(\alpha(\bar{g}) \cup \text{Pre}_{ap}(R, n))
\]

where \( \text{Ifp} \) is a least fixed point iteration [3] which starts with \( R = (0, \ldots, 0) \), and on each iteration joins the current approximate set with the approximate successor set. Finally after reaching convergence, it returns a tuple \( R \) to \( \text{FwdReach}_{ap}(q_0) \) or \( \text{BackReach}_{ap}(\bar{g}) \) as the case may be. The approximate set of states that can be reached is the implicit conjunction \( \gamma(\text{FwdReach}_{ap}(q_0)) \). The approximate set of states that can reach \( \bar{g} \) is is the implicit conjunction \( \gamma(\text{BackReach}_{ap}(\bar{g})) \).

Using Lemma 1 and monotonicity of \( \text{Im} \) and \( \text{Pre} \) functions, it can be shown that the derived functions \( \text{Im}_{ap} \) and \( \text{Pre}_{ap} \) have the property

\[
\text{Im}(R(x), n) \subseteq \text{Im}(\gamma(\alpha(R(x))), n) \subseteq \gamma(\text{Im}_{ap}(\alpha(R(x)), n))
\]

\[
\text{Pre}(R(x), n) \subseteq \text{Pre}(\gamma(\alpha(R(x))), n) \subseteq \gamma(\text{Pre}_{ap}(\alpha(R(x)), n))
\]

The proof that \( \text{FwdReach}_{ap} \) (and \( \text{BackReach}_{ap} \)) are overapproximations (supersets) follows trivially. These operators give us exact results in the special case when there is just one subset, \( w_1 = x \), in the collection \( w \).

4 Overlapping Projections

Our scheme for choosing the collection of subsets is presently manual. Of course, it would be desirable to automate, fully or partially, the choice of
subsets and we are working on developing good heuristics to do so. Our present heuristic [10] tries to put interacting finite state machines (FSMs) together in one subset. Often a master FSM communicates with a number of other slave FSMs. This is captured by having blocks, where the master is paired with each of its slaves in different blocks. Occasionally two rather big state machines have a small interface, which can be captured by adding bits through which the two machines communicate to the subsets having the corresponding FSMs.

4.1 Computing $\text{Im}_{ap}$ by Multiple Constrain

The key step in our approximate forward propagation is computing $\text{Im}_{ap}$.

$$\text{Im}_{ap} (R, n) = (S_1, \ldots, S_p) = \alpha (\text{Im}(\gamma (R), n(x, y)))$$

We would like to be able to compute the $S_j$'s separately, without computing $\text{Im}(\gamma (R), n)$. Clearly $S_j$ can only depend on the next state functions of the variables appearing in the $j^{th}$ block, $w_j$ in $W$. Let $\alpha_j(n)$ be the subset of predicates determining the next state for the bits in $w_j$. Clearly, $S_j = \text{Im}(\gamma (R), \alpha_j(n))$.

To avoid unnecessary BDD blowup, we want to avoid the explicit conjunction $\gamma (R)$. $S_j$ can be computed, by forming the next state relation for block $w_j$ and using early quantification [3]. However this did not work when we tried it on our larger examples. Instead Condert and Madre [6] have shown how to compute the image of a Boolean function vector, using the generalized cofactor (also called constrain) operator $(\downarrow)$. $(f \downarrow g)(x)$ has the same value as $f(x)$ when $g(x)$ holds, and usually results in a smaller BDD than $f$.

Condert and Madre [6] show that $\text{Im}(\gamma (R), \alpha_j(n)) = \text{Im}(1, \alpha_j(n) \downarrow \gamma (R))$. To avoid computing the large BDD for $\gamma (R)$, it is tempting to compute $\alpha_j(n) \downarrow R_1 \downarrow R_2 \ldots \downarrow R_p$. This works [15] well if the supports of $R_i$'s are disjoint. However since we have overlapping subsets, the naive method is incorrect [10].

Instead, for overlapping projections, we use the method of multiple constrain [10]. Let $(z_1, \ldots, z_p)$ be dummy state bits with corresponding next state functions $(R_1, \ldots, R_p)$. The multiple constrain method relies on the following key observation

$$\text{Im}(\gamma (R_1, \ldots, R_p), \alpha_j(n)) = \text{Im}(1, [\alpha_j(n), R_1, \ldots, R_p]) \downarrow z_1 \downarrow z_2 \ldots \downarrow z_p$$

We can optimize on the usual recursive co-domain partitioning algorithm [6], by avoiding computing the parts of the range that will be discarded. The algorithm $\text{Im}_{mc}$ described below implements the required function $\text{Im}_{ap}$. (A more detailed treatment is given in [10]).

function $\text{Im}_{mc} ((R_1, \ldots, R_p), (n_1, \ldots, n_m))$
$v \leftarrow [n_1, \ldots, n_m, R_1, \ldots, R_p]$
for $j=p$ down to 1 by 1 do
$v \leftarrow v \downarrow v[m + j]$
endfor
return $\text{Im}(1, \{v[1], \ldots, v[m]\})$
5 Using Auxiliary Variables to refine $Im_{ap}$ and $Pre_{ap}$

The previous schemes can be further improved upon 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. 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 use auxiliary state variables [12] to broaden applicability of approximate reachability techniques.

5.1 Converting Internal Wires to Auxiliary State Variable

We look for important internal conditions in the combinational logic and convert them to auxiliary variables. An auxiliary 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.

We make use of auxiliary variables by converting them to state variables. To assign a next state function to an auxiliary variable, we get the fanin cone for the internal wire it corresponds to. (A fanin cone of a wire is obtained by topologically moving back from the wire and grabbing all the logic that feeds to it until we hit a flop boundary or an input boundary). Let $f(x)$ be the Boolean function for cone of logic feeding into a wire, called $foo$. Recall that $n$ is the next state functions for the usual state variables $x$. The next state function for auxiliary state variable $foo$ is obtained by substituting the corresponding next state function from $n$ for each state variable in the support of $f(x)$. This has the effect of retiming the internal wire $foo$. (The initial condition for auxiliary state variable $foo$ is set by the image computation $Im(q_0, f)$). This construction is possible for only those internal wires whose fanin cones involve just state variables and no inputs.

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 $M = \langle x, y, q_0, n \rangle$, can be transformed to $M' = \langle x', y', q'_0, n' \rangle$, where $x' = x \cup y$ and $q'_0 = q_0$. 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, it is the corresponding input variable from $y'$. Assuming totally unconstrained input environment, $M$ and $M'$ allow the same externally visible behaviors. However $M'$ allows us more flexibility in choosing 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 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. We presently look for internal wires in the RTL description that have many state variables in their fanin support. More details on our heuristic can be obtained from [12].

6 Refinement

An overapproximation of the states that lie on a path from the initial state \( q_0 \) to a state not satisfying a user-specified property \( g \) is computed by repeated forward and backwards passes, until the approximation no longer improves.

function BackAndForth \( (g) \)
\[ R_f \leftarrow (0, \ldots, 0) \]
\[ R_b \leftarrow (1, \ldots, 1) \]
while \( (R_f \neq R_b) \) do
\[ R_f \leftarrow \text{lfp } R.(\alpha(q_0) \cup (\text{Im}_{ap}(R, n) \cap R_b)) \]
if \( (\gamma(R_f) \rightarrow g) \) return “no errors”
\[ R_b \leftarrow \text{lfp } R.(\alpha(\hat{g}) \cup (\text{Pre}_{ap}(R, n) \cap R_f)) \]
if \( (\gamma(R_b) \wedge q_0 = 0) \) return “no errors”
endwhile
return \( R_f \)

The tests \( \gamma(R_f) \rightarrow g \) and \( \gamma(R_b) \wedge q_0 = 0 \) can be performed without computing the explicit conjunctions of the BDDs in \( R_f \) and \( R_b \) by computing images, using the method of multiple constrain [10]. \( \gamma(R_f) \rightarrow g \) holds iff \( \text{Im}(\gamma(R), g) = \{1\} \) and \( (\gamma(R) \wedge q_0) = 0 \) iff \( \text{Im}(\gamma(R), q_0) = \{0\} \). If BackAndForth is unable to prove the desired property \( g \), it is often possible to run it again with larger blocks of variables in \( w \).

6.1 Counterexamples

If BackAndForth reports a possible error, it is useful to check whether there is an actual error by generating an example path from \( q_0 \) to a state that does not satisfy \( g \). This both confirms the existence of an error and provides debugging information to the user. In exact reachability analysis, if an error state is reachable from an initial state, it is straightforward to construct a specific path from the initial state to an error. But in approximate analysis, such a path may not exist. More subtly, the algorithm may have found a real error via a non-existent path. A simple search method was implemented for counterexample generation which worked well on examples.

Starting from the error states, the algorithm computes approximate preimages and stores the preimages obtained at the various iterations of the fixpoint algorithm in a stack. Let \( T_0, T_1, \ldots, T_m \) (where \( T_m \) intersects with the error states) be the final contents of the stack, and let \( T_1 \) be the first level at which the approximate preimage intersects with the initial state \( q_0 \). Choose a single state, \( s_0 \) from the intersection \( q_0 \wedge T_1 \) and compute an exact image of \( s_0 \). If the image of \( s_0 \) intersects with \( T_{i+1} \), choose a single state \( s_i \) from the intersection
and continue moving forward. It is also possible that the image of some state 
$s_i$ in layer $T_j$ may lie entirely in $T_j$ and not intersect with $T_{j+1}$ at all (implying 
$T_{j+1}$ is approximtely reachable from $s_i$ but not exactly reachable from $s_i$),
in which case, randomly choose another state $s_{i+1}$ from the image of $s_i$ and 
continue trying to move to the next layer in the stack. If the algorithm spends 
more than 10 steps at the same layer, it aborts and reports that it could not find 
a counterexample.

This simple algorithm has worked well on proving local safety properties 
over the individual submodules of FLASH I/O, but often fails when we prove 
global safety properties over the complete design. We are currently working 
on improving this and looking for ways to improve the approximations when 
the counterexample generation gets stuck.

7 Experiments

The experimental implementation of the method was in LISP, calling David 
Long’s BDD package (implemented in C) via the foreign function interface. 
The method was evaluated on a collection of control circuits from the MAGIC 
chip, a custom node controller in the Stanford FLASH multiprocessor [13]. 
For comparison with earlier work, we also present our results when applied to 
the ISCAS89 benchmark suite.

Approximate Forward Reachability: In the case of s13207 circuit from the 
ISCAS-89 benchmark suite, earlier approximate schemes based on disjoint 
partitions [5] resulted in a superset with a satisfying fraction of 3.42e-106, 
whereas our scheme with overlapping projections resulted in a tighter superset 
with a satisfying fraction of 1.13e-115, which represents an improvement by 
3.3e+08. Similarly in case of s38584, results with overlapping projections 
were better by a factor of 8.8e+15. A more detailed listing of the results 
we obtained on the other circuits from the ISCAS89 suite and the results on 
the FLASH I/O modules is given in [10]. Further on adding auxiliary state 
variables the results obtained by overlapping projections over the usual state 
variables alone, was further improved by at least an order of magnitude. More 
details on the results obtained with auxiliary state variables are in [12].

Approximate Forward and Backward Reachability: We applied our approxi-
mate forward and backward routines to prove some designer provided invariant 
properties on various submodules in FLASH I/O. Out of 20 properties, the 
approximation scheme was able to prove 13 of them, and present counterex-
amples for the remaining 7. (More details on the results with the modules in 
FLASH I/O can be obtained from [11]).

Proving global properties on a big design: We have also applied our al-
gorithm to prove some more global properties over FLASH I/O. Using the 
lossless cone-of-influence reduction, we are able to reduce the original design 
(nearly 2400 state variables) to the order of 200 state variables. By doing ap-
proximate reachability over these 200 variables using overlapping projections, 
we have been able to prove 3 global invariants and disprove 2 others with a 
valid counterexample. However there is still more to be done before designs
of this size can be directly handled by our model checker.

References


