Verification by Approximate Forward and Backward Reachability [*]

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

Abstract

Approximate reachability techniques trade off accuracy for the capacity to deal with bigger designs. In this paper, we extend the idea of approximations using overlapping projections to symbolic backward *reachability. This is combined with a previous method of computing overapproximate forward reachable state sets using overlapping projections. The algorithm computes a superset of the set of states that lie on a path from the initial state to a state that violates a specified invariant property. If this set is empty, there is no possibility of violating the invariant. If this set is non-empty, it may be possible to prove the existence of such a path by searching for a counter-example. A simple heuristic is given, which seems to work well in practice, for generating a counter-example path from this approximation. We evaluate these new algorithms by applying them to several control modules from the I/O unit in the Stanford FLASH Multiprocessor.*

1 Introduction

Binary Decision Diagrams (BDDs) [1] have enabled formal verification to tackle 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.

We are interested in 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 the 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.

Of course, overapproximations are subject to *false errors.* We call states that violate a user specified property *bad states.* Even if there is a bad state in S_{ap} S might not contain a bad state. However, knowing that S_{ap} contains a bad state may still be useful. First, perhaps the approximation can be improved to show that there are actually no bad states. Second, it is often possible to search efficiently for a state in S that violates the property, by limiting the search to S_{ap}. The existence of such a state conclusively proves that the property may be violated. Finally, it may be possible to modify the design to ensure that S_{ap} (of the new design) contains no bad states, which would allow verification of the property while potentially making the design more robust.

The method described here successively refines an approximation of the set of states that lie on a path from the initial state to a bad state. It alternates forward and backward passes; each pass uses the approximation computed by the previous pass (called the current set, below). A forward pass finds a subset of the current set which appears to be reachable from an initial state, while a backward pass finds a subset of the current set which seems to consist of predecessors of the bad states. It repeats until the set of states no longer changes (i.e., until a fixed point is reached).

If the resulting set is empty, no bad states are reachable from the initial state, so the property has been verified. Otherwise, the method has not ruled out the possibility of there being a path from the initial state to a bad state (and hence, a design error). So a simple heuristic is used that computes a *subset* of the reachable states from the initial state that is likely to contain the bad state. If this succeeds, the error is a genuine error and a counterexample path can be reported to the user for debugging.

The approximation used is based on *overlapping projections* of sets of states, represented as binary decision diagrams (BDDs). 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 method 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 (the violated assertions re-

[*]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.

sulted from omitting constraints on the possible inputs to the design).

Algorithms for efficiently computing the set of projections of the states reachable from an initial state were described in DAC98 [7]. This work extends that by adding efficient backwards traversal from the bad states, along with a search for definite errors when an error cannot be ruled out by the overapproximation. The most difficult aspect of this problem is efficiently computing a sufficiently accurate preimage of an implicit conjunction of BDDs. The implementation is based on cofactoring on the values of the domain variables.

1.1 Related Work

At a high level, this work is quite similar to that of Wong-Toi, et al. [5], 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* [10, 11] use "high density" BDDs to compute an *underapproximation* of the forward reachabe set. Cho *et al* [4] 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* [3] combine approximate forward reachability with *exact* backward reachability. Lee *et al* [9] 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 [4], [3] and [9], we allow for overlapping subsets, as overlapping projections have been shown [7] to be a more refined approximation compared to earlier schemes based on disjoint partitions.

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. 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 $[x \to \mathcal{B}]$, where $\mathcal{B} = \{0,1\}$. The initial state $q_0 \in [x \to \mathcal{B}]$. The next state function is $\mathbf{n} : [x \to \mathcal{B}] \times [y \to \mathcal{B}] \to [x \to \mathcal{B}]$.

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 [2]. For example, let $R(x)$ be a predicate with support in x, we can compute the image of R under $\mathbf{n}$ as

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

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

$$Pre(\bar{g}, \mathbf{n}) = \lambda x.\exists x', y.(x' = \mathbf{n}(x, y)) \wedge \bar{g}(x').$$

2.1 Approximation by Projections

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 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 α_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 $R \to \alpha_j(R)$. The projection operator α projects a predicate $R(x)$ onto the various w_j's, and its associated concretization operator γ conjoins the collection of projections.

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

Lemma 1 *For every predicate $R(x)$ and collection of subsets $(w_1, \ldots, w_p)$ of x, $R \to \gamma(\alpha(R))$.*

Let $\mathbf{R} = (R_1, \ldots, R_p)$ and $\mathbf{S} = (S_1, \ldots, S_p)$ be two equally sized tuples. We define the *meet* ($\sqcap$) and *join* ($\sqcup$) operator between $\mathbf{R}$ and $\mathbf{S}$ as follows:

$$\begin{aligned} (R_1, \ldots, R_p) \sqcap (S_1, \ldots, S_p) &= (R_1 \wedge S_1, \ldots, R_p \wedge S_p) \\ (R_1, \ldots, R_p) \sqcup (S_1, \ldots, S_p) &= (R_1 \vee S_1, \ldots, R_p \vee S_p) \end{aligned}$$

Note that $\gamma(\mathbf{R}) \cup \gamma(\mathbf{S}) \subseteq \gamma(\mathbf{R} \sqcup \mathbf{S})$. Hence the join operator is an approximation of set union. (However, the meet operator is an exact set intersection operator, since $\gamma(\mathbf{R}) \cap \gamma(\mathbf{S}) = \gamma(\mathbf{R} \sqcap \mathbf{S})$).

The operator α 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. 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. Let Im_{ap} (the subscript *ap* denotes "approximate") return the projected version of the image of an *implicit* conjunction of BDDs.

$$Im_{ap}(\mathbf{R}, \mathbf{n}) = \alpha(Im(\gamma(\mathbf{R}), \mathbf{n}(x, y)))$$

Using Im_{ap}, we can compute an overapproximation, $FwdReach_{ap}(q_0)$, of the reachable states for a machine M as follows:

$$FwdReach_{ap}(q_0) = \text{ lfp } \mathbf{R}.(\alpha(q_0) \sqcup Im_{ap}(\mathbf{R}, \mathbf{n}))$$

where lfp is a least fixed point iteration [2] which starts with $\mathbf{R} = (0, \ldots, 0)$, and on each iteration *joins* the current approximate set with the approximate successor set. Finally after reaching convergence, it returns a tuple $\mathbf{R}$ to $FwdReach_{ap}(q_0)$. The overapproximate reachable states set is the *implicit* conjunction $\gamma(FwdReach_{ap}(q_0))$.

Similarly, let Pre_{ap} return the projected version of the preimage of an *implicit* conjunction of BDDs.

$$Pre_{ap}(\mathbf{R}, \mathbf{n}) = \alpha(Pre(\gamma(\mathbf{R}), \mathbf{n}(x, y)))$$

Using Pre_{ap}, we can compute an overapproximation, $BackReach_{ap}(\bar{g})$, of the set of states in M that can reach the set of states $\bar{g}$ as follows:

$$BackReach_{ap}(\bar{g}) = \text{ lfp } \mathbf{R}.(\alpha(\bar{g}) \sqcup Pre_{ap}(\mathbf{R}, \mathbf{n}))$$

As in the forward case, the least fixed point routine above starts with $\mathbf{R} = (0, \ldots, 0)$ and on reaching convergence, it returns a tuple $\mathbf{R}$ to $BackReach_{ap}(\bar{g})$. The overapproximated set of states that can reach $\bar{g}$ is is the *implicit* conjunction $\gamma(BackReach_{ap}(\bar{g}))$.

$FwdReach_{ap}$ (and $BackReach_{ap}$) are overapproximations because the image (and preimage) at every iteration of the least fixpoint routine is an overapproximation. These approximate operators give us exact results in the special case when there is just one subset, $w_1 = x$, in the collection w.

3 Overlapping Projections

Recently, it has been shown [7] that overlapping projections are an improved approximation scheme compared to earlier schemes based on disjoint partitions. Further an efficient algorithm to compute the function $Im_{ap}(\mathbf{R}, \mathbf{n})$ was proposed in DAC98 [7]. In this paper we propose an efficient algorithm to compute the function $Pre_{ap}(\mathbf{R}, \mathbf{n})$.

3.1 Computing Pre_{ap} by Domain Cofactoring

The key step in symbolic backward propagation algorithms is the preimage computation.

$$Pre_{ap}(\mathbf{R}, \mathbf{n}) = (S_1, \ldots, S_p) = \alpha(Pre(\gamma(\mathbf{R}), \mathbf{n}(x, y)))$$

Instead of using next state *relations* to compute the preimage [2, 9], Filkorn [6] showed that the the preimage of a set represented by a BDD Q, can be obtained by substituting the state variables in Q with their corresponding next state function.The obvious algorithm to compute S_j would be to substitute the functions in $\gamma(\mathbf{R})$ and then hide existentially all the variables apart from those appearing in w_j. However, since most of the variables would be hidden, the size of the intermediate BDD during this computation would be prohibitive even when the final BDD was small.

Instead, S_j is computed by recursively cofactoring on the domain variables in w_j, which allows the existential quantification to be done on the fly. Each state variable x in $\mathbf{R}$ is renamed to x' to avoid conflicts. Let σ be a map from each x'_i to the function that is to be substituted for it. Initially, σ maps x'_i to its next state function, but σ is modified in the recursive calls to the preimage function. Only some of the functions in σ will be used because some x'_i variables do not appear in any R_i; let $|\sigma|$ be the number of functions in σ that will actually be substituted.

The recursive algorithm Pre_{dc} (the subscript dc denotes "domain cofactoring") takes as arguments the current substitution, σ, the current approximation $\mathbf{R}$, the approximate reachability set from the first forward pass $\mathbf{I}$, and the set of variables w_j to project onto. $\mathbf{I}$ is used to prune preimage states that are definitely not reachable. (The algorithm shown below to compute S_j, assumes there are only two subsets in our collection $\mathbf{w}$. The extension to any arbitrary number of subsets is obvious. We use $\downarrow$ to denote the ordinary cofactor operator).

```
function $Pre_{dc}(\sigma, [R_1, R_2], [I_1, I_2], w_j)$
    if $(I_1 == 0)$ or $(I_2 == 0)$ return $0$
    if $(|\sigma| == 0)$ return $R_1 \wedge R_2$
    $v \leftarrow$ next variable to cofactor on
    $t \leftarrow Pre_{dc}(\sigma \downarrow_v, [R_1 \downarrow_v, R_2 \downarrow_v], [I_1 \downarrow_v, I_2 \downarrow_v], w_j)$
    $e \leftarrow Pre_{dc}(\sigma \downarrow_{\bar{v}}, [R_1 \downarrow_{\bar{v}}, R_2 \downarrow_{\bar{v}}], [I_1 \downarrow_{\bar{v}}, I_2 \downarrow_{\bar{v}}], w_j)$
    if $(v \in w_j)$  result $\leftarrow ite(v, t, e)$
    else  result $\leftarrow (t \vee e)$
return result
```

The following optimizations are used to reduce the number of recursive calls to Pre_{dc}:

- If any of the BDDs in $\mathbf{I}$ becomes 0 during cofactoring, the algorithm immediately returns 0. This computes an *on-the-fly* conjunction of the approximate preimage with the invariant.
- The substitution σ only includes functions that need to be substituted into the R_i's. Further, if at any point the support of a function in σ is wholly contained inside w_j, it is immediately substituted into the R_i's and thereafter removed from σ. When $|\sigma| = 0$, all the the support of all R_i's is contained in w_j, so the algorithm computes their explicit conjunction and returns.
- The algorithm cofactors only on the variables in w_j. However it doesn't cofactor on variables from w_j that don't appear in σ (it may be necessary to quantify them existentially in $\mathbf{I}$, though).
- After cofactoring on variables in w_j, the support of the functions in σ is disjoint from w_j, and now the result of Pre_{dc} is either 0 or 1. Since, by this point in the recursion, the BDDs are generally small, the algorithm does the substitution and returns 1 only if the resulting BDD is not a constant 0. This approach worked fine on all the examples that were tested; however, in case of BDD blowup, the algorithm could return a conservative value of 1.

3.2 Refinement

Because each pass is approximate, repeated forward and backwards passes yield progressively more accurate results. Each step of each forward and backward traversal is intersected with the set of states computed by the previous traversals. Passes are alternated until the approximation no longer improves. Here is the verification algorithm, which computes 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.

```
function $BackAndForth\ (g)$
$\mathbf{R_f} \leftarrow (0, \ldots, 0)$
$\mathbf{R_b} \leftarrow (1, \ldots, 1)$
while $(\mathbf{R_f} \neq \mathbf{R_b})$ do
    $\mathbf{R_f} \leftarrow lfp\ \mathbf{R}.(\alpha(q_0) \sqcup (Im_{ap}(\mathbf{R}, \mathbf{n}) \sqcap \mathbf{R_b}))$
    if $(\gamma(\mathbf{R_f}) \rightarrow g)$ return "no errors"
    $\mathbf{R_b} \leftarrow lfp\ \mathbf{R}.(\alpha(\bar{g}) \sqcup (Pre_{ap}(\mathbf{R}, \mathbf{n}) \sqcap \mathbf{R_f}))$
    if $(\gamma(\mathbf{R_b}) \wedge q_0 = 0)$ return "no errors"
endwhile
return $\mathbf{R_f}$
```

The tests $\gamma(\mathbf{R_f}) \rightarrow g$ and $\gamma(\mathbf{R_b}) \wedge q_0 = 0$ can be performed without computing the explicit conjunctions of the BDDs in $\mathbf{R_f}$ and $\mathbf{R_b}$ by computing images, using the method of multiple constrain [7]. $\gamma(\mathbf{R_f}) \rightarrow g$ holds *iff* $Im(\gamma(\mathbf{R}), g) = \{1\}$, and $\gamma(\mathbf{R}) \wedge q_0) = 0$ *iff* $Im(\gamma(\mathbf{R}), q_0) = \{0\}$. 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, beginning with an initial approximation derived from the previous result. This is possible because restricting to the previous approximate sets greatly reduces the size of the BDDs during the approximation.

3.3 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_i 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 $q_0 \wedge T_i$ and compute an exact image of s_0. If the image of s_0 intersects with T_{i+1}, choose a single state s_1 from the intersection and continue moving forward. It is also possible that the image of some state s_l in layer T_j may lie entirely in T_j and not intersect with T_{j+1} at all (implying T_{j+1} is approximately reachable from s_l but not exactly reachable from s_l), in which case, randomly choose another state s_{l+1} from the image of s_l and continue trying to move to the next layer in the stack. Thus, unlike the conventional method of generating counter-examples from exact preimages, we may have to spend more than one step at the same 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 in practice, and has generated counterexamples in all cases where the algorithm could not prove the desired property.

4 Experiments

The method was evaluated on a collection of control circuits from the MAGIC chip, a custom node controller in the Stanford FLASH multiprocessor [8]. The circuits are control intensive; the state bits do not include data path bits. Table 1 gives a brief description of the various control modules in the I/O unit.

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

Module	State Bits	Input Bits
IOInboxQCtl	23	8
ReqDecode	37	27
ReqService	41	58
IOMiscBusCtl	44	18
PciInterface	88	55

The experimental implementation of the method was in LISP, calling David Long's BDD package (implemented in C) via the foreign function interface. The properties to prove were invariants provided by the designer. (Traditional benchmarks, such as ISCAS 89, do not come with specified properties, so they could not be used here.) The maximum number of BDD nodes was limited to 10 million nodes for each experiment. The variable subsets w were chosen manually, using the same heuristics as in [7].

4.1 Results

In the tables below, *Inv* lists the property to be proved. The column under P gives the results of the verification effort. A 'Y' means that property was proved, 'N' means a counter-example was generated, and '?' means that the verification exercise could not be completed.

Nodes is the maximum number of BDD nodes that existed at a time during the experiment, and *Time* is the cpu time (in seconds) to complete the experiment on a MIPS R4300 with 768MB main memory (the cpu time includes time spent during Lisp garbage collection).

The *Exact* column shows results of the exact preimages of the error states, when it was possible to compute them. The exact preimages were computed relative to the approximate reachable set computed during the first forward pass of the approximate algorithm. The same variable ordering was used in all the examples, to get the numbers for the *Exact* method and the *Approximate* method.

Table 2. IOInboxQCtl Invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	4,216	9.5	Y	4,196	10.8
p2	Y	4,408	9.5	Y	4,312	10.8
p3	N	112,257	80.6	N	75,600	88.0
p4	Y	5,519	9.5	Y	4,850	10.8
p5	N	119,710	81.0	N	79,619	86.4

Table 3. ReqDecode Invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	97,362	52.2	Y	42,954	49.9
p2	Y	680,107	76.1	Y	88,213	59.5

Table 4. ReqService Invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	95598	517.8	Y	74419	419.1
p2	N	121573	1276.7	N	93799	860.4
p3	Y	94510	820.0	Y	74419	418.5
p4	Y	112367	1021.6	Y	94365	418.5

4.2 Discussion

The approximate scheme is able to prove or disprove the property in all the cases, unlike the exact method which fails to complete the verification exercise for most of the properties in the *PciInterface* design example. Further, the approximate approach uses fewer BDD nodes to prove or disprove the invariant. The difference in the required number of BDD nodes is fairly large, in most of the cases. Note that in case of the *PciInterface* design example, the approximate method completes the verification exercise well within the 10 million node limit.

For the smaller example of *IOInboxQCtl*, the approximate method takes marginally more time than the exact method. The time advantage of the approximate method becomes clearer as we go for the larger design examples. Most of the time was spent in the approximate forward traversal (which was done for both the *Exact* and *Approximate* case).

The input environment for these design examples was assumed to be totally non-deterministic. The "errors" reported here were all because of such an overly general environment model. Extension to this work would be to incorporate better environment models.

Table 5. IOMiscBusCtl Invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	N	2936929	1031.5	N	512469	301.5
p2	Y	1791385	850.4	Y	426324	302.3

Table 6. PciInterface Invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	?	>10 mil	?	Y	1012742	559.8
p2	Y	1116686	2271.2	Y	1007843	661.6
p3	?	>10 mil	?	Y	1324916	750.9
p4	?	>10 mil	?	N	2060485	1290.6
p5	?	>10 mil	?	N	1268233	686.9
p6	?	>10 mil	?	N	2097440	973.6
p7	Y	1113254	468.8	Y	1007408	420.1

5 Acknowledgments

We thank Hema Kapadia, the chief designer of the I/O unit in the MAGIC chip, for providing invariants for the examples, and Jules Bergmann for helping us use his tool, *vex*, as a front end Verilog parser.

References

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

[2] 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 1990*, pp. 428-439.

[3] Cabodi, G., Camurati, P., and Quer, S., "Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals," *EURO-DAC 1994*, pp. 22-27, 1994.

[4] Cho, H. et. al, "Algorithms for Approximate FSM Traversal Based on State Space Decomposition," *IEEE TCAD*, Vol. 15, No. 12, pp. 1465-1478, December 1996.

[5] Dill, D. L., and Wong-Toi, H., "Verification of Real-Time Systems by Successive Over and Under Approximation," *CAV 1995*, pp. 409-422.

[6] Filkorn, T, "Functional Extension of Symbolic Model Checking," *CAV 1991*, pp. 225-232.

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

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

[9] Lee, W., Pardo, A., Jang, J., Hachtel, G., and Somenzi, F., "Tearing Based Automatic Abstraction for CTL Model Checking," *ICCAD 1996*, pp. 76-81.

[10] Ravi, K., and Somenzi, F. "High-density Reachability Analysis," *ICCAD 1995*, pp. 154-158.

[11] Ravi, K., McMillan, K. L., Shiple, T. R., and Somenzi, F., "Approximation and Decomposition of Binary Decision Diagrams," *DAC 1998*, pp. 445-450.