An Iterative Method for Reached State Approximation

Cliff Cordeiro

Abstract

In this paper we present a new method for computing the approximate reached state set of a finite state machine by modifying its transition relation. Both upper and lower bounds are computed using this method.

1 Introduction

Computing the set of reached states in a *finite state machine* (FSM) is a key operation in both verification [1] and synthesis [2] & [3]. The reached state set allows designers to check a design for errors and other unwanted behavior. It is a precise measure of the exact set of states a design can take under all possible inputs.

Given a directed graph, and a set of initial nodes in the graph, reachability computes the set of nodes on some path from the initial nodes. A FSM can be represented by a directed graph, which is also called a *state transition graph*. Computing the reachable states (nodes) of this graph can be done by a breadth first search traversal of the state transition graph beginning at the initial states.

Unfortunately this computation explodes when the number of states in the FSM becomes very large, as is the case in real designs. This is often called the *state explosion problem*. To attempt to overcome this problem, an implicit representation called a binary decision diagram or BDD [4], is used to represent the required quantities. e.g. the transition relation (TR), which implicitly represents the FSM's state transition graph, and the initial and reachable sets of states.

In order to further combat the state explosion problem, we propose to perform modifications on the transition relation before computing the reached state set. This will have the effect of simplifying the computation while introducing some error. By limiting the types of modifications to the TR, we can assure that the value returned by the reachability computation is either an upper or a lower bound. By performing intelligent modifications we hope to achieve adequate bounds on the reached state set without the computation exploding.

The paper is organized as follows: the basic definitions are given in section 2. Previous work is described in section 3. In section 4 we present the algorithm used to modify the transition relation BDDs. Some results are given and discussed in section 6. We conclude with future directions for this work.

2 Definitions

Definition 1 Finite State Machine: *A finite state machine* $\mathcal{M}$ *is a 5-tuple* $(Q, \Sigma, \Gamma, T, I)$ *where*

- Q *is a finite set of states*
- Σ *is a finite set of input values*
- Γ *is a finite set of output values*
- $T \subset Q \times \Sigma \times \Gamma \times Q$ *is the transition relation*
- I *is a set of initial or starting states of the machine.*

$T(q, \sigma, \gamma, t) = 1$ means that from state $q \in Q$ on input $\sigma \in \Sigma$, there is a transition to some state $t \in Q$, while the output is $\gamma \in \Gamma$. Thus an FSM can be represented by a *state transition graph*, whose vertices are states, and edges are labelled with elements of $(\Sigma \times \Gamma)$.

Definition 2 Reachable states*: The set of reachable states is denoted by R, $q \in R$ if and only if there is a path from some initial state $q_0 \in I$ (the set of initial states) to q.*

In general, let x represent the present state variables and y represent the next state variables. $T(x, y)$ represents the transition relation, which defines a relationship between present states (x variables) and next states (y variables) in the state transition graph, irrespective of input and output, and $T(y, x)$ represents the same transition relation, with the caveat that the x and y variables are interchanged (i.e. y used for present state). Let $R(x)$ denote the reachable states, and $I(x)$ denote the initial states. In general, since x, y are fixed (present, next state) variables over the state space, any relation $r(x, y)$, gives rise to a new relation $r(y, x)$ by interchanging the x and y variables.

Definition 3 Hamming Distance Operator*:The Hamming distance operator, when given the complete set of v variables, w temporary variables, and a range upon those variables, returns a BDD which represents the function that evaluates to* true *when the value of the v variables is a distance one from the value of the w variables in the given range. That is, the variables are allowed to differ only in the given range, and in that range, only by a Hamming distance of one (or one bit). It is then possible to use this BDD to convert a set of possible v values into a set of values on identical variables a constrained distance one away using elementary BDD operations. The $dist(u, w)$ BDD can be computed using:*
$dist(u, w) = \sum_{i=0..n}(u_i \oplus w_i)\prod_{j \neq i}(u_j \overline{\oplus} w_j)$
Since $dist(u, w)$ is a symmetric function, the BDD is compact.

3 Previous work

Computing the set of reachable states in the transition relation of a finite state machine is equivalent to performing a traversal of the state transition graph, beginning at the initial states. This traversal may be breadth first or depth first.

Touati et-al [5] Burch et-al [6], and Coudert et-al [7] independently extended this concept to handle reachability in large systems by using implicit methods. All quantities (transition relations, sets of states etc.) are represented by BDD's (binary decision diagrams), and the algorithm is represented by a fixed point computation.

Algorithm Reached($T(x, y), I(x)$)
  $R^0(x) = I(x)$
  While $R^n(x) \neq R^{n+1}(x)$
    $\exists_y T(y, x) R^n(y) + R^n(x)$
  return $R(x)$

Unfortunately, this algorithm still has a tendency to blow up with large designs because the transition relation becomes unweildy and traversal time explodes. A solution to this problem is to compute only an approximation of the reached state set. Ravi et-al [8] have illustrated a method for computing a bound on the reached state set by modifying R(x) at each iteration. This decreases the size of R(x) and allows the computation to continue in order to reach more states. This method can only compute a lower bound on the reached state set though, in the limit, the actual set will be computed. Also, by refusing to modify the TR, they are not gaining much speed in the image computation phase of the algorithm, and are not simplifying the actual process as much as possible. In very large examples, where the initial image computation itself is very costly, this algorithm will fail.

4 Our Approach

We propose to modify the transition relation BDD of the FSM in order to achieve reachability computation. The primary purpose of modifying the transition relation is to shrink the BDD representing it, which will, in turn, increase the speed of the traversal to compute the reached state set.

BDDs are unique objects in that their size is not completely determined by the amount of minterms they represent. Any large BDD can be reduced in size by either adding or deleting minterms.

It can be shown that adding edges to the TR can only add states to the reachable state set, while subtracting edges can only subtract states from the set. If the edge to be added connects two reached states, or an unreached state to a reached state, there is no effect on the result of reachabilty. If the edge to be added connects a reached state to an unreached state, however, the unreached state will be included in the upper bound. If the edge to be deleted connects an unreachable state to any other state, there will be no effect on the result of reachabilty. However, if the edge to be deleted is the only transition to a reached state, that reached state and possibly its neighbors will be removed from the lower bound.

4.1 Adding Edges

A heuristic was necessary in order to decide which edges were beneficial to add to the TR. In future applications, it would be good to try to add those edges which have no effect on the reached state set in order to compute an exact solution. This is not the approach taken here. Edges were added/subtracted solely to shrink the BDD, in order to compute bounds. This approach had the advantage of decoupling the BDD operations from the meaning of the transition relation. That is, the concentration was on the BDD and not what it represented.

The heuristic used for addition held that minterms a small Hamming distance from the minterms contained within the BDD would likely shrink the BDD. Many options were attempted in order to find the best set of close minterms to add. It was possible to choose minterms differing only in a small window of bits, minterms of distance one or greater, etc. Refer to section 2 for the definitions and section 6 for the options available.

The procedure used was to take the constraints given by the user to create a BDD representing the $dist(x, y)$ function on the variables. This BDD was then used to generate the minterms the given distance from the TR onset, these were then added to the TR. If a distance greater than one was required, this procedure was repeated. A concise formation of this algorithm follows, with $depth$ being the number of iterations to take, $C(x)$ being the transition relation, and $dist(x, y)$ being the BDD returned by the Hamming Distance operator (section 2).

Algorithm($Addnodes(depth, C(x), dist(x, y))$**)**
  $i = 0$
  While $i < depth$
    $C(x) = \exists_y dist(x, y)C(x) + C(x)$
    $i++$
  return $C(x)$

4.2 Removing Edges

The heuristic used for subtraction held that minterms a small Hamming distance from the minterms in the main body of the offset of a BDD would likely shrink that BDD. This is analogous to saying that those minterms far from the onset should be deleted. It was also possible to constrain this computation to a window of variables of a distance of one or greater.

The procedure for computing lower bound transition relation BDDs used the same $dist(x, y)$ function as the upper bound computation. This BDD was used to pull out all possible minterms close to the offset. These were then inverted and anded with the TR in order to subtract them from the BDD. The following algorithm summarizes this computation.

Algorithm($Subnodes(depth, C(x), dist(x, y))$**)**
  $i = 0$
  $D(x) = !C(x)$
  While $i < depth$

$D(x) = \exists_y dist(x, y) D(x) + D(x)$
$i++$
$C(x) = C(x)!D(x)$
return $C(x)$

4.3 Computing the Set

For both the upper and lower bound TRs, an additional step was added in order to assure a smaller BDD was generated. To compute the function with the (heuristically) smallest BDD that lies between some upper bound $f + d$ and lower bound f, the generalized cofactor fd may be used, as in [5]. This function was used to compute intermediate BDDs between our calculated TR BBDs and the actual TR BBD.

The modified BDDs are then supplied to a function which computes the exact reached state set. If the modifications are good, the lower and upper bounds are close together and are computed in less time then the exact set would take.

4.4 Classes of Changes

An additional approach to computing good bounds relies on merging bounds computed with different methods, or classes of changes, to the TR. The aim is to compute disjoint bounds and to combine them to create very good bounds. The combination of these bounds is trivial.

The combination of upper bounds is achieved by taking the intersection of the various sets, or the Boolean and of their BDDs. This is because each upper bound is guaranteed to contain the actual state set, and taking the intersection will remove only those states not common to all the bounds. The combination of lower bounds is achieved by taking the union of the various sets, of the Boolean or of their BDDs. This is because every state contained in the lower bounds is guaranteed to be in the actual state set, so taking the union will capture all actual states in any of the lower bounds. With good classes of changes, which pick up relatively disjoint bounds, a good approximation of the actual state set can be made. In general :

- $R_{upper} = R_{u1} \bigcap R_{u2} \ldots R_{um}$
- $R_{lower} = R_{l1} \bigcup R_{l2} \ldots R_{lm}$

where ui's refer to upper bound change sets and li refer to lower bound change sets.

5 Implementation

These functions were implemented on top of the Verification Interacting with Synthesis (VIS) system currently being developed by the University of California at Berkeley and the University of Colorado at Boulder. VIS provides formal verification and fast simulation of a finite state system specified in any of various formats, including HDLs. Our project is implemented in VIS.

Although previously, the TR has been conceived of as a single large BDD, in actuality, VIS implements the TR as a graph of communicating FSMs. This method is more efficient in terms of storage as well as speed of computation. Our approach had to be modified in order to work on a network of FSMs. We modified it in the obvious fashion, by traversing the FSMs within the product machine and modifying each TR separately.

6 Results

While implementing the code, many additional options were found to be helpful in creating better bounds on the actual reached state set. These are enumerated in the next section.

There were three options used to limit the window of variables considered during the distance one operation, which controls the node add/subtract operations. Those options were -l, -u, and -w. -l and -u were used to put lower and upper bounds on the variable numbers, which were allowed to differ by a Hamming distance of one from the on or off set (depending on the bound desired.) Either could be used independently so that the opposite bound would be the first or last variable as the case might be. -w was used to explicitly specify a window of a given size at the end of the variable set. While it is possible to create a window of those variables with the -l option, it is not known a priori how many variables are in the TRs so that a window of this type can't be specified without -w for all partitioned BDDs.

It was seen early on that it is counter-productive to modify those TRs in the product machine that are small to begin with. The -n $\sharp$ option was added to rectify this. If a product BDD has a support of less than $\sharp$ variables, it will be left alone, speeding up the modify step of computation while assuring good bounds. This turned out to have a significant effect on both computation time and bound quality.

To compute worse bounds, but in a shorter time, the depth of edges to add/subtract to/from the TR can be increased arbitrarily using -d. This means that edges will be added/subtracted, then the set of close/far edges will be computed again until the give n depth is reached.

It is also possible to compute either the lower bound, upper bound, the actual reached state set, or a combination of these using other options.

Finally, another option, -c, is given which allows the user to choose the number of classes of changes to try. The classes are selected internally using -l and -u distributed evenly over the variable space in order to add edges which differ in the selected variables. For example, with 6 variables and 3 classes, the algorithm would compute bounds by looking for minterms differing in only the first two variables, then the second two, and finally the last two. It would then combine the bounds computed using the algorithm given above.

6.1 Varying the Parameters

On the following section, the effects of changing -l, -w, -n and -c on four of the ISCAS examples are given. A brief summary of the effects follows.

Figure 1 shows the effect of choosing a window of variables at the lower end of the BDD order to constrain the distance one function. When the window is small, the bounds are close to the actual set, however, increasing the size of the window has an adverse effect on the quality of the bounds.

Figure 2 shows the effect of choosing a window of variables at the top of the BDD order to constrain the distance one function. Again, the bounds worsen as the window is enlarged, however, in general the bounds aren't as good as the bottom of tree bounds.

Figure 3 shows the effect of the -n option, which ignores those component TRs with a support of less than the given number of variables. This has a good effect on the bounds; however, with -n too large, the bound computations become the same as the actual computation because no changes are made on the TR as a whole.

Figure 4 shows the effect of the -c option, which takes different classes of changes and merges them to compute better bounds. This actually returned better and better bounds with increased -c, although they were not monotonically better due to the method used to choose the classes. I.e., choosing 8 classes might have hit upon a good class of changes to make while increasing that to 9 would miss that class by grouping the variables in a different fashion.

6.2 Benchmark Examples

Figure 5 shows the results of a generic run on a variety of small FSMs using the options -d 1 -n 10 -w 3. The upper bounds were decent in most cases, though the lower bounds need improvement. Changing the options or taking more classes of changes would bring the bounds closer together as the previous graphs showed. The smaller examples had upper, lower, and actual bounds equal due to the effect of the -n option.

Figure 6 illustrates the time taken on the computations in the previous figure. It shows that in most cases, the bound computations are faster than the actual computations, even when taking into account the time to modify the TR. In the few cases were the bound computation took longer than the actual computation, the actual compuation itself was quick, and modifying the TR just wasted time. It is not particularly useful to compute bounds on small examples. These were just used to show the benefits and correctness of the method.

7 Conclusions and Future Work

As the examples in the previous section show, some options have a better effect on the bounds than others. It became clear that the -n (ignore small product transition relation BDDs) option was necessary, though the correct value depends on the size of the example. It was also shown that -u (window at the end) was a much better change than -w (window at the beginning), though both returned non-trivial bounds when the window size was small enough. The simplest way to compute the best bounds seems to be to use a large value of -c (large number of classes), which, though time consuming, returns better and better bounds as its parameter value is increased.

It was also seen that the lower bounds for many examples were unacceptable. This was partially fixed by the application of the -c and -n parameters. However, it has become obvious that the lower bound and upper bound computations have significant differences in their accuracy given a set of parameters. It should be possible, and actually is in the latest version of the code, to calculate the lower bounds and upper bounds independently of each other. This allows the bounds to be tweaked independently.

On the other hand, it is for another project to show that sharing the information produced by each bound with the other will be beneficial in computing a good, fast approximation. By informational theory alone it should be more efficient to share information between the computations. This can also be seen empirically.

It is known that the states in a lower bound are guaranteed to be actual reached states. It is known that all of the reached states are contained within the upper bound. Modifying the transition relation with this in mind can be done in such a way that the set of reached states is unaffected, yet the size of the TR BDD may be significantly reduced. All transitions from any states to the states in a lower bound may be safely added to the TR in order to shrink it. As well, all transitions from those states not contained in the upper bound (actual unreachable states) to any other states may be safely added to the TR. This computation may be done iteratively until the entire reached state set is computed.

References

[1] H. Touati, R. K. Brayton, and R. P. Kurshan, "Checking Language Containment using BDDs," in *Proc. of Intl. Workshop on Formal Methods in VLSI Design*, (Miami, FL), Jan. 1990.

[2] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. Intl. Conf. on Computer Design*, pp. 328–333, Oct. 1992.

[3] H. Cho, G. D. Hachtel, and F. Somenzi, "Redundancy Identification and Removal Based on Implicit State Enumeration," in *Proc. Intl. Conf. on Computer Design*, pp. 77–80, Oct. 1991.

[4] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[5] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 130–133, Nov. 1990.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[7] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 126–129, Nov. 1990.

[8] K. Ravi and F. Somenzi, "High Density Reachability Analysis," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 154–157, Nov. 1995.