DECIDING STATE REACHABILITY FOR LARGE FSMs

by

Thomas R. Shiple, Rajeev K. Ranjan,
Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton

Memorandum No. UCB/ERL M97/73

15 August 1997

DECIDING STATE REACHABILITY FOR LARGE FSMs

by

Thomas R. Shiple, Rajeev K. Ranjan,
Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton

Memorandum No. UCB/ERL M97/73

15 August 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Deciding State Reachability for Large FSMs

Thomas R. Shiple* Rajeev K. Ranjan
Alberto L. Sangiovanni-Vincentelli Robert K. Brayton
Department of EECS, University of California, Berkeley, CA 94720

Abstract

We address the state reachability problem in FSMs, which is the problem of determining if one set of states can reach another. State reachability has broad applications in formal verification, synthesis, and testing of synchronous circuits. This work attacks this problem by making a series of under- and over-approximations to the state transition graph, using the over-approximations to guide the search in the under-approximations for a potential path from one state set to the other. Central to this method is an algorithm to approximate a Boolean function by another function having a smaller BDD.

*Supported by an SRC Fellowship

Contents

1 Introduction

We are concerned with the problem of determining if there exists a path, in the state transition graph of a system of interacting FSMs, from a given set of initial states I to a final set F. We call this the *state reachability problem.* This problem is more specific than the usual problem of determining the set of states R reachable from I. Obviously, if R is known, then by checking if R and F intersect, the state reachability problem can be answered. However, it may not be necessary to compute R to decide state reachability.

Finding efficient algorithms to solve state reachability is crucial because several problems in verification, logic synthesis and testing can be efficiently reduced to the state reachability problem. For example, to determine if two FSMs M_1 and M_2 are equivalent, we can define the set of initial states I to be those product states that are initial in both M_1 and M_2, and F to be those products states where the output values of M_1 and M_2 differ. Then, M_1 and M_2 are equivalent if and only if I cannot reach F in the product of M_1 and M_2. As another example, checking safety properties specified as automata over finite strings can be reduced to state reachability. In particular, if a monitor T is defined that enters a BAD state when a property is violated, then an FSM M satisfies the property if and only if the BAD states cannot be reached from the initial states, in the product of M and T. In summary, an efficient solution to the state reachability problem would provide efficient solutions to a host of other CAD problems.

When an FSM is described as a set of interacting FSMs, the state reachability problem is PSPACE-complete [1]. Even so, algorithms based on symbolic breadth-first traversal using BDDs can handle FSMs with several hundred flip-flops. However, on larger examples, the standard approaches start to falter because

1. the BDD representing the set of states reached at an intermediate step grows too large, or
2. the image of a given set of states cannot be computed.

This work does not address the first problem directly, but instead focuses on the second problem; in doing so, we aim to increase the size of FSMs that can be analyzed.

To understand the idea behind our approach, consider the state transition graph G of an FSM M, representing a set of interacting FSMs. We assume that G is too large to build and analyze directly. Instead, we make a series of over- and under-approximations to G, where with each approximation, we attempt to narrow in on a path from I to F, or prove that such a path cannot exist. An over-approximation of G is a graph containing a superset of the edges[1] of G, and an under-approximation of G is a graph containing a subset.

Consider an over-approximation V to G, and restrict V to those transitions lying on a path from I to F. If there is a path in G from I to F, then this path must exist in the restricted V. Now, consider an under-approximation U. Denote by I' all those states that are reachable from I in U, and by F' all those states that can reach F in U. If I' and F' intersect, then certainly I can

[1]The terms *edge* and *transition* are used interchangeably.

reach F in G, because, by definition, all of the transitions in U are in G. On the other hand, if I' and F' do not intersect, then we try to extend the frontier of I' by looking for true transitions (i.e., those in G) among those in V that lead from states in I'. If no such transition can be found in V, then we have proven that I cannot reach F in G. We also try to work backwards from F' at the same time, in a similar manner. In summary, V is used to guide the search in U for a path from I to F.

The feasibility of our approach is predicated upon finding approximations to G that have reasonable BDD sizes, but yet are close enough approximations to G to permit useful information to be derived. To this end, we introduce two new BDD operators, called *bddOverApprox* and *bddUnderApprox*. Consider the BDD F representing a function f. *BddUnderApprox* selectively replaces some subgraphs in F by the constant ZERO, yielding a new BDD G representing the function g, such that G has fewer nodes than F, and $g \subseteq f$. Nodes are selected for replacement based on a cost function that takes a parameter that controls the tradeoff between reducing the BDD node count, and reducing the onset size. In a similar manner, *bddOverApprox* adds minterms to a function by replacing BDD nodes by the constant ONE. Ravi and Somenzi [17] independently and concurrently formulated the same BDD approximation problem, although they apply it in a different situation.

The main contribution of this work is a BDD-based algorithm to solve the state reachability problem, via a series of under- and over-approximations to the state transition graph. To our knowledge, the idea of using both under- and over- approximations in this domain is novel. Also, we define, and give a heuristic solution to, the new problem of approximating Boolean functions to yield a small BDD.

2 The state reachability problem

An FSM is specified as a 5-tuple, $M = \langle S, I, \mathcal{X}, \Sigma_I, T \rangle$, where:

- S is the state space of size 2^l, spanned by the binary variables $x = [x_1, x_2, \ldots, x_l]$. We also introduce a second set of variables $y = [y_1, y_2, \ldots, y_l]$ to denote the next state.
- I is a subset of S, denoting the initial states.
- $\mathcal{X}$ is the set of n inputs, with associated binary variables $u = [u_1, u_2, \ldots, u_n]$.
- $\Sigma_I = B^n$ is the input alphabet.
- T is the next state function, $T : S \times \Sigma_I \rightarrow S$. T is presented as a vector of l Boolean functions, $\delta = [\delta_1, \delta_2, \ldots, \delta_l]$, where δ_i is the next state function of the state variable x_i. $T_i = (y_i \equiv \delta_i(u, x))$ gives the corresponding transition relation of x_i, and $T = \prod_{i=1}^{l} T_i$. δ is typically given as a multi-level logic network.

The *state transition graph* $G(x, y)$ of M is a binary relation over S defined by

$$G(x, y) = \exists u \prod_{i=1}^{l} T_i(x, u, y_i).$$

That is, $G(x,y) = 1$ if and only if there exists an input u such that from state $x = [x_1, x_2, \ldots, x_l]$, δ_i evaluates to y_i, for all $1 \leq i \leq l$. The state sequence $\pi = x^0 x^1 \ldots$ is a *run* of M if $(x^i, x^{i+1}) \in G$, for all $i \geq 0$. The run π is *initialized* if $x^0 \in I$.

Definition 1 *An instance of the* state reachability problem *consists of an FSM* $M = \langle S, I, \mathcal{X}, \Sigma_I, T \rangle$ *and a subset of states* $F \subseteq S$. *The answer to the state reachability problem is YES if there exists an initialized run* $x^0 x^1 \ldots x^r$ *in the state transition graph of* M *such that* $x^r \in F$, *and NO otherwise.*

As defined, the next state behavior of an FSM must be deterministic. However, sometimes FSMs are specified with nondeterminism, meaning that for a given input and present state, there may be more than one next state. In this case, a relation, rather than a function, is needed to specify the next state behavior. Since we are only interested in the state transition graph G derived from the relation T, nondeterminism does not pose a problem; we stick with determinism for clarity.

Finally, we are interested in analyzing a system of interacting machines, but our definition of the problem is with respect to a single FSM. This limitation is easily overcome by realizing that any system of FSMs can be thought of as a single FSM by amassing the next state functions of all the state variables of the system into a single vector of next state functions.

3 Related work

The problem of traversing the state graph of an FSM, whether to compute the set of reachable states or to determine if a specific subset of states is reachable, has been the object of intensive research over the last decade. A breakthrough occurred in 1989 when Coudert, Berthet and Madre proposed using BDDs to perform symbolic breadth-first traversal of state graphs [9]. With this approach, the number of states in the graph is no longer the principal limitation (as in depth-first traversal); instead the "complexity" of the Boolean functions defining the underlying circuit govern the efficiency.

Since symbolic traversal was proposed, many researchers have suggested various heuristics in a quest to traverse ever larger and complex FSMs. These heuristics have been shown to be effective in some cases, and not so in others. Here we review some of the previous work, and how it relates to our research.

3.1 Image computation

Image computation is the central task in symbolic traversal. Given a set of states A, we want to determine the successors of A in the graph G of an FSM. The image can be computed as

$$image(y) = \exists x (G(x,y) \cdot A(x)).$$

That is, a state y is a successor of A if there exists a state x in A such that there is an edge from x to y in G. Substituting for G from above,

$$image(y) = \exists x \exists u (\prod_{i=1}^{l} T_i(x, u, y_i) \cdot A(x)).$$

The naive approach of first taking the product of the T_i's and A, and then quantifying x and u, is ill-conceived because often the intermediate product is large even though the final result is not so large. In general, the full product must be computed because existential quantification does not distribute over Boolean conjunction. However, a special case can be exploited where it does distribute. Namely, the equation

$$\exists x (f(x, y) \cdot g(y))$$

can be rewritten as

$$\exists x (f(x, y)) \cdot g(y).$$

Several researchers have used this fact to quantify some variables before the entire product is formed, in an attempt to avoid the intermediate blowup in the overall computation [3, 11, 12, 16, 19].

Another technique for simplifying image computation is to use certain sets of states as don't cares to simplify the BDDs of the set A of states and the individual transition relations T_i. In particular, suppose that the set R of states has already been reached, and during the previous image computation, the set B was reached for the first time. For the next image computation step, there is no harm in re-exploring states in R that are not in B. Thus, any set C such that $B \subseteq C \subseteq R$ is suitable for the next image computation. In addition, we can arbitrarily choose the behavior of T_i on any state x not in C, since such states are disregarded when the eventual product with C is formed. The operators *constrain* and *restrict* implement the simplification of BDDs using don't care sets [8, 19, 18].

Cabodi *et al.* [4] introduce the *existsCofactor*, which is similar to the constrain operator, but allows existential quantification to distribute over conjunction. In particular, they are able to rewrite

$$\exists x (f(x, y) \cdot g(x, y))$$

as

$$\exists x f(x, y) \cdot \exists x g'(x, y)),$$

where $g'(x, y)$ is the *existsCofactor* of g with respect to f.

All three of the techniques discussed in this subsection are orthogonal to our approach, and in fact can be used in combination with our approach

3.2 Exact state reachability

Balarin introduced an algorithm to test for language emptiness of automata over infinite strings [2]. For ease of presentation, we describe his algorithm for the simpler case of automata over finite strings. An automaton over finite strings has a designated set F of accepting, or final, states. The

language of such an automaton is not empty if and only if there exists a path from I (the initial states) to F; thus, state reachability provides the answer to language emptiness.

To solve this problem, Balarin makes a series of successively finer over-approximations to the state graph G of the automaton, in an attempt to determine if I can reach F. Each over-approximation V is analyzed to determine if there is a path from I to F. There are two cases to consider.

1. There is a path from I to F in V: If this path exists in G (G is analyzed for this single path), then the language is not empty, and the algorithm terminates. Otherwise, a new over-approximation is constructed that eliminates this path in V, and the procedure is repeated.
2. There is not a path from I to F in V: Then the language is empty, and the algorithm terminates.

Balarin's algorithm partly served as inspiration for our approach. However, whereas Balarin analyzes a single path from I to F in V at a given iteration, we analyze all of the paths from I to F to guide the search for a real path in the under-approximation U.

The work of Cabodi *et al.* [5] is similar in spirit to ours. They first compute an over-approximation of the states reachable from I, and then use this information to constrain an exact backward search from F. If I is reached in the backward search from F, the state reachability problem is answered in the affirmative.

The work of Courcoubetis *et al.* [10] addresses the problem of not being able to build the transition relation for an FSM M. Instead of employing BDDs and performing BFS on the state graph, they use DFS, building up the graph one transition at a time. Since this method is explicit, they are limited in time to exploring roughly 10^8 states. This method is referred to as "on-the-fly", because states are checked to see if they belong to F *while* the graph is being built.

3.3 Approximate state reachability

The set of *reachable states* is the set of states R that can be reached from the initial states I. Obviously, if R is known, then the set I can reach F if and only if R and F have a non-empty intersection. Even if R is not known, sometimes an approximation to R can be used to answer the state reachability problem. If an under-approximation R^- and F intersect, then clearly R and F intersect, and the answer to state reachability is "YES". On the other hand, if an over-approximation R^+ and F do not intersect, then R and F do not intersect, and the answer is "NO". In the cases where R^+ or R^- cannot be used to answer the state reachability problem, they could be used as a starting point to focus the search for a path from I to F.

Cho *et al.* [6] have devised various techniques for over-approximating R. The first step of these techniques is to partition the flip-flops of the FSM to yield a set of k interacting sub-FSMs. This partitioning is done so that flip-flops with strong interaction tend to be placed in the same sub-FSM. Next, the set R_i of reachable states of sub-FSM$_i$ is computed for each i. This computation

does not consider the full, dynamic interaction of sub-FSM$_i$ with the other sub-FSMs, but instead considers some partial constraints on the values of the inputs of sub-FSM$_i$ driven by the other sub-FSMs. This yields an over-approximation of the state reachable in sub-FSM$_i$. Finally, an over-approximation R^+ of the entire reached set is given by the Cartesian product $R^+ = R_1 \times \ldots \times R_k$. They use the complement of R^+ as an under-approximation of the unreachable states to perform logic minimization.

Ravi and Somenzi [17] propose a technique to under-approximate R. Their algorithm proceeds with the usual symbolic BFS, but when the set of states A to explore becomes too large (in terms of BDD size), they continue the search from only a subset of A. This subset is heuristically chosen to have a small BDD while retaining as many states as possible from A. As mentioned previously, this problem of approximating a set using a small BDD is the same problem we have formulated. As such, we could employ the heuristics proposed by Ravi and Somenzi in our work.

As mentioned earlier, Courcoubetis *et al.* [10] perform state graph traversal using depth-first traversal, and not using BDDs. To represent the set of states visited thus far in a traversal, they use a data structure whose size is directly proportional to the size of the set of states. The problem is that this set may become too large to represent. To combat this problem, they use a hash table without collision chains, which hashes a state to a single bit indicating if the state has already been visited. Since collisions may occur in this hash table, it may be incorrectly deduced that a state has already been visited, when in fact it has not been. This may in turn lead to visiting only a subset of the total set of reachable states, thus yielding an under-approximation. They use probabilistic analysis to quantify the probability of collisions for a randomly chosen hash function.

4 Algorithm to decide state reachability

Figure 1 gives an outline of our algorithm to solve the state reachability problem for the FSM M and final state set F. The algorithm also takes $0 \leq \alpha \leq 1$ as input, which controls the degree of approximation used during the algorithm. Here we discuss the top-level control of the algorithm and illustrate the algorithm in detail with an example. In Section 6, we discuss each of the subroutines in detail.

We assume that the graph $G(x, y)$ of M is too big to build and manipulate. Instead, we construct a series of approximations to $G(x, y)$ that are small enough to manipulate efficiently. We are willing to trade off execution time in favor of memory savings, in an attempt to handle very large FSMs.

The initial step is to compute an over-approximation $V(x, y) \supseteq G(x, y)$ (line 1), and an under-approximation $U(x, y) \subseteq G(x, y)$ (line 2). G is a set of directed edges of a graph, so V is a superset and U is a subset of this set of edges. The goal is to choose $V(x, y)$ and $U(x, y)$ so that they are close to $G(x, y)$, but have much smaller representations.

The algorithm then iterates, adding edges to $U(x, y)$ and removing edges from $V(x, y)$, until it is determined whether or not there exists a path from I to F in G. The search is conducted working forward from I and backward from F. In particular, each iteration starts by performing,

```
reachable(M, F, α)

1     V := InitialOverApprox(M, α)
2     U := InitialUnderApprox(M, α)

3     I := I of M
4     F := input parameter F

5     while (TRUE)
6       I := I + ForwReachability(U, I)  /* states reachable from I in U */
7       F := F + BackReachability(U, F)  /* states which can reach F in U  */

8       if (I intersects F)
9         return "yes, I can reach F"

10      V := RestrictToFinal(V, I, F)  /* restrict V to edges on paths from I to F  */

11      if (I cannot reach F in V)
12        return "no, I cannot reach F"

13      toF := V(x,y) · ¬F(x) · F(y)  /* transitions to F in V  */
14      fromI := V(x,y) · I(x) · ¬I(y)  /* transitions from I in V  */

15      falseToF(x,y) := ApproxFalseEdges(M, toF, V, α)  /* subset of false edges in toF  */
16      trueToF(x,y) := ApproxTrueEdges(M, toF, V, α)  /* subset of true edges in toF  */
17      falseFromI(x,y) := ApproxFalseEdges(M, fromI, V, α)  /* subset of false edges in fromI  */
18      trueFromI(x,y) := ApproxTrueEdges(M, fromI, V, α)  /* subset of true edges in fromI  */

19      if (falseToF and trueToF and falseFromI and trueFromI are empty)
20        trueToF(x,y) := ExactTrueEdges(M, toF)
21        trueFromI(x,y) := ExactTrueEdges(M, fromI)
22        if (trueToF or trueFromI is empty)
23          return "no, I cannot reach F"
24        else
25          falseToF(x,y) := toF - trueToF(x,y)
26          falseFromI(x,y) := fromI - trueFromI(x,y)

27      V := V - (falseToF + falseFromI)  /* remove false edges from V  */
28      U := U + (trueToF + trueToI)  /* add true edges to U  */
```

Figure 1: Algorithm to decide state reachability.

in U, forward reachability from I and backward reachability from F (lines 6 and 7). Reachability is carried to a fixed point. Any states reached from I are added to I, and any states that can reach F are added to F. Since all edges in U are present in G, all states in I and F can also be reached in G. If at any time I and F intersect, then we know that a path exists from the original I to the original F in G (lines 8 and 9).

The next step is to restrict V to those edges that lie on some path from I to F in V (line 10). If there does indeed exist a path from I to F in G, then it must lie in the restricted V. Hence, if it is discovered that there is no path from I to F in V, then we can immediately conclude that no such path exists in G (lines 11 and 12). This restriction is done on each iteration, because as we will see, we eliminate some edges from V on each iteration (line 27). Remember that we assume that the representation for V is small enough so that we can do reachability on V efficiently.

Since we carried reachability in U to a fixed point (lines 6 and 7), by definition, there are no edges in U leaving I or entering F. Hence, to continue the search in U, we need to find edges of G leaving I and edges entering F. Where do we look for such edges? Naturally, we look for them in V. In particular, V focuses our search for a path from I to F, since if such a path exists in G, it must exist in V. Furthermore, V contains *only* those edges lying on a path from I to F in V (because of line 10). This is a key point. Thus, in lines 13 and 14, we restrict V to those edges entering F (*toF*) and those exiting I (*fromI*). Our intention is to determine which edges in these sets are "true" (exist in G) and which are "false" (do not exist in G). Just answering the question for one edge is already NP-complete (reduction from SAT). Hence, we try to approximate the true and false sets (lines 15-18). If all of the approximations are empty, then we must do *exact* analysis (lines 20 and 21) on the true edges in order to draw any conclusions. If exact analysis does not find any true edges, then we can conclude (line 23) that no path exists from I to F in G.

If at least one of *falseToF*, *trueToF*, *falseFromI*, and *trueFromI* is non-empty, then we remove false edges from V, and add true edges to U (lines 27 and 28). Removing false edges from V is important because it may further narrow the search. At this point, we repeat the entire loop.

There are three major subroutines in this procedure:

1. initial over-approximation of G (line 1),
2. initial under-approximation of G (line 2), and
3. approximation of edges from I and to F (lines 15-18).

Each of these subroutines involves approximating a set of edges represented by a BDD. For this task, we make extensive use of the BDD approximation operators. Section 5 addresses the BDD approximation problem.

4.1 Example

We illustrate the algorithm with a detailed example. Figure 2 shows the graph $G(x, y)$ of a 21-state FSM. The states are labeled for reference. The set of initial states is $I = \{3\}$, and the set of final

states is $F = \{13\}$. To emphasize, we do not have direct access to this graph in the algorithm; it is shown here to clarify the operation of the algorithm.

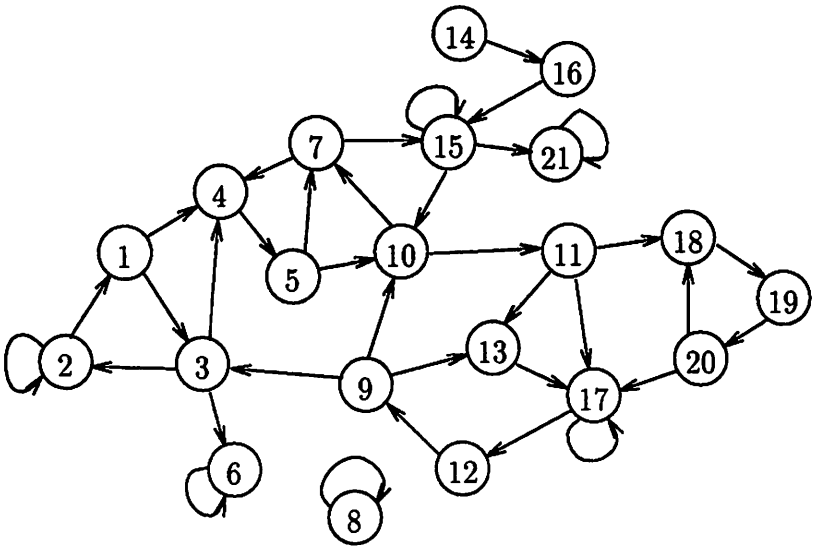

Figure 2: Graph G of a 21-state FSM. The initial state is 3 and the final state is 13.

The initial over-approximation V_1 and under-approximation U_1 are shown in Figures 3 and 4, respectively. Throughout the example, the current sets of states I and F are each indicated by a dotted region. In addition, the false edges of the over-approximations (although not known a priori by the algorithm) are indicated by dots.

Iteration 1 Performing reachability in U_1 (lines 6, 7), I is enlarged to $\{2,3\}$ (because of the edge $3\rightarrow 2$) and F is enlarged to $\{9,13\}$ (because of $9\rightarrow 13$). Since $I \cap F = \emptyset$, we proceed to restrict V_1 (line 10) to edges on paths from I to F, to yield V_1' in Figure 5.[2] For example, edges $3\rightarrow 6$ and $16\rightarrow 15$ are removed. With careful analysis, even edges $9\rightarrow 3$ and $1\rightarrow 3$ can be removed, since their removal does not affect the reachability question. In addition, we remove the self-loops in V_1. Since I cannot reach F in V_1', the condition at line 11 if false.

As a side note, as the algorithm is presented, whenever we can remove edges from V_1, we do so. In reality, removing such edges may make the BDD for V needlessly larger. Hence, these edges should be treated as don't cares and used to minimize the BDD size of V. In addition, edges between two states in I, or between two states in F, can be treated as don't cares in both V and U.

Continuing with the algorithm, line 13 identifies those edges of V_1' entering F,

$$toF = \{5\rightarrow 9, 10\rightarrow 13, 11\rightarrow 13, 12\rightarrow 9\}.$$

and line 14 identifies the edges of V_1' leaving I,

$$fromI = \{2\rightarrow 1, 3\rightarrow 4, 3\rightarrow 5\}.$$

[2]Now the new I and F are shown. Edges contained within I or within F are not drawn.

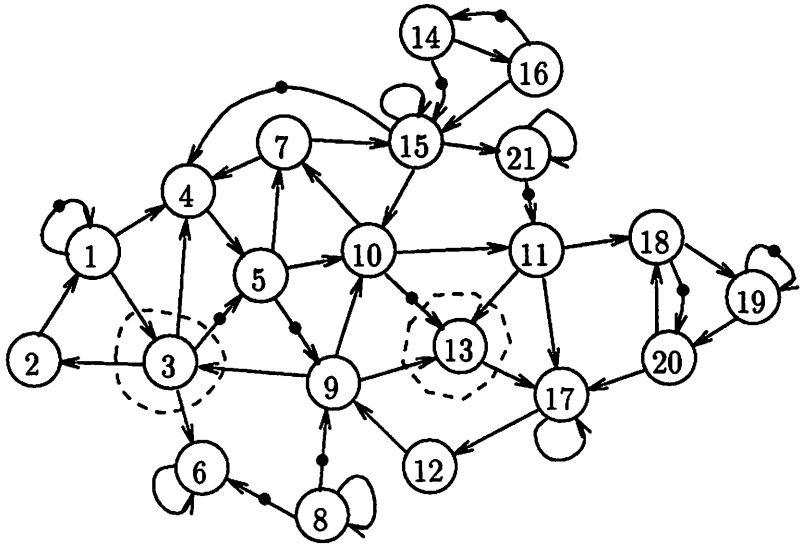

Figure 3: Initial over-approximation V_1. The sets I and F are indicated by the dotted region. False edges are indicated by a dot.

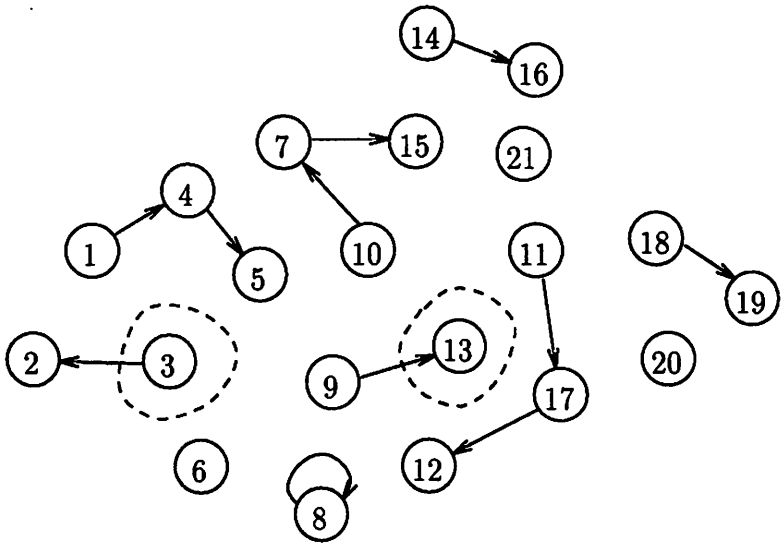

Figure 4: Initial under-approximation U_1.

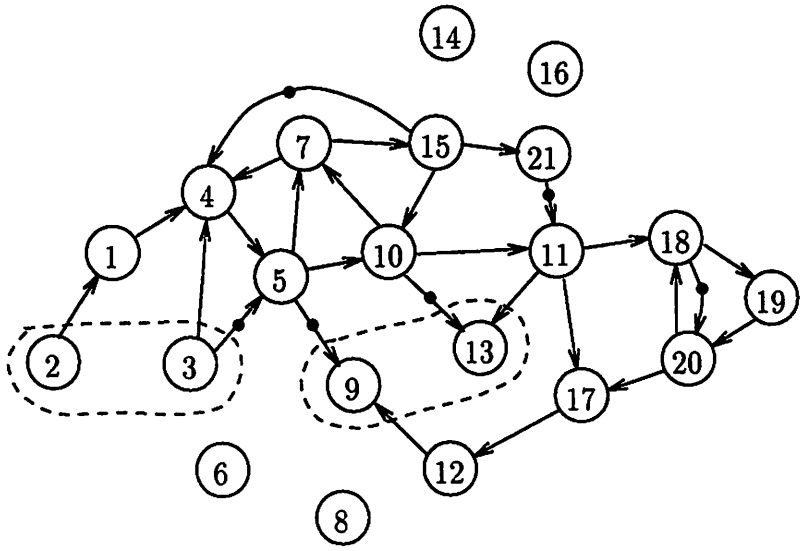

Figure 5: Over-approximation V_1', formed by restricting V_1 to edges on paths from I to F. Edges contained within I or within F are not drawn.

Since these edges belong to an over-approximation, we do not know a priori which edges are true and which are false. Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} \mathit{falseToF} &= \emptyset, \\ \mathit{trueToF} &= \{12{\rightarrow}9\}, \\ \mathit{falseFromI} &= \{3{\rightarrow}4\}, \\ \mathit{trueFromI} &= \{2{\rightarrow}1\}. \end{aligned}$$

Since not all of these are empty, the algorithm proceeds to lines 27 and 28, where edge $3{\rightarrow}4$ is removed from V_1' to yield V_2 (Figure 6), and edges $12{\rightarrow}9$ and $2{\rightarrow}1$ are added to U_1 to yield U_2 (Figure 7).

Iteration 2 Reachability on U_2 expands I to $\{1,2,3,4,5\}$, and expands F to $\{9,11,12,13,17\}$. Notice that pre-existing edges in the under-approximation (e.g., $1{\rightarrow}4$, $4{\rightarrow}5$) allow the reachability computation to progress beyond those edges just added to the under-approximation.

I and F still do not intersect, so we restrict V_2 (for example, by removing all edges to and from states 18, 19 and 20), to yield V_2' (Figure 8). I can still reach F in V_2' (line 11), so we compute

$$\begin{aligned} \mathit{toF} &= \{5{\rightarrow}9, 10{\rightarrow}13, 10{\rightarrow}11, 21{\rightarrow}11\}, \text{ and} \\ \mathit{fromI} &= \{5{\rightarrow}7, 5{\rightarrow}9, 5{\rightarrow}10\}. \end{aligned}$$

Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} \mathit{falseToF} &= \{10{\rightarrow}13, 21{\rightarrow}11\}, \\ \mathit{trueToF} &= \emptyset, \end{aligned}$$

$$falseFromI = \{5 \rightarrow 9\},$$
$$trueFromI = \{5 \rightarrow 7\}.$$

Then the false edges are removed from V_2' to yield V_3 (Figure 9), and the true edge is added to U_2 to yield U_3 (Figure 10).

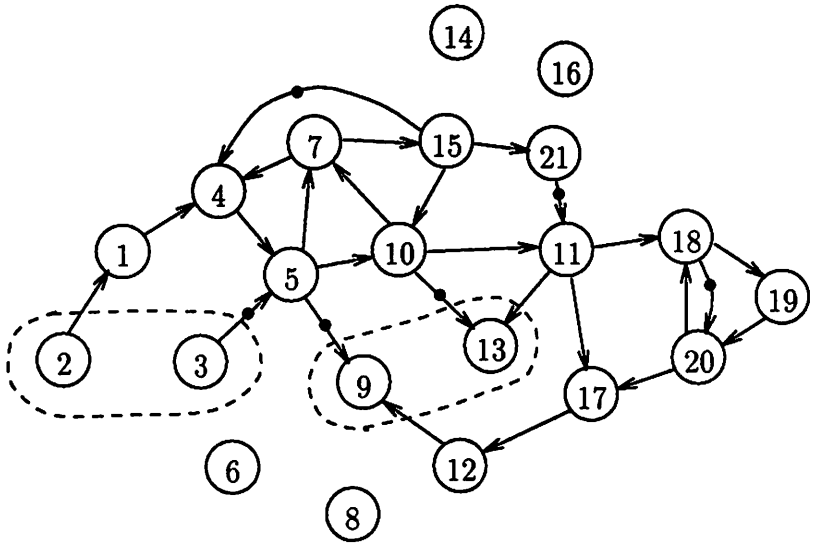

Figure 6: Over-approximation V_2, formed by removing edge 3→4 from V_1'.

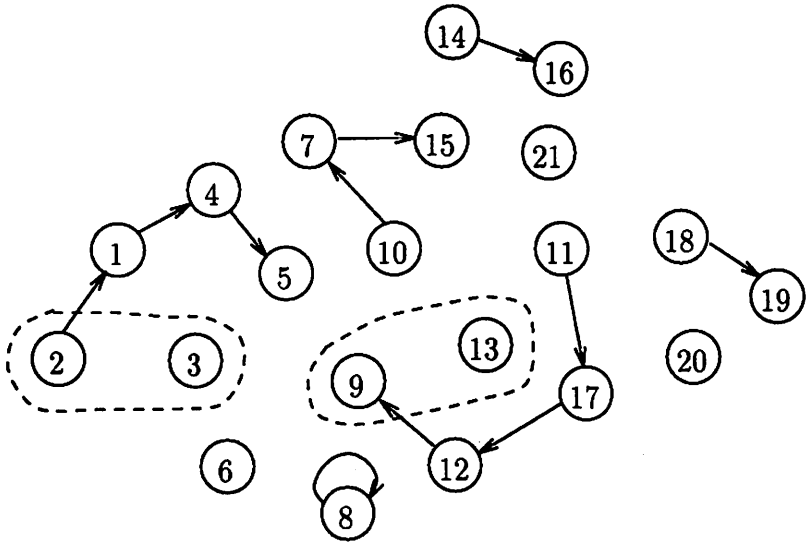

Figure 7: Under-approximation U_2, formed by adding edges 12→9 and 2→1 to U_1.

Iteration 3 Reachability on U_3 adds states 7 and 15 to I. I and F do not intersect. Restricting V_3 removes edges 10→7 and 15→21, to yield V_3' (Figure 11). Attention is focused on

$$toF = \{10 \rightarrow 11\}, \text{ and}$$

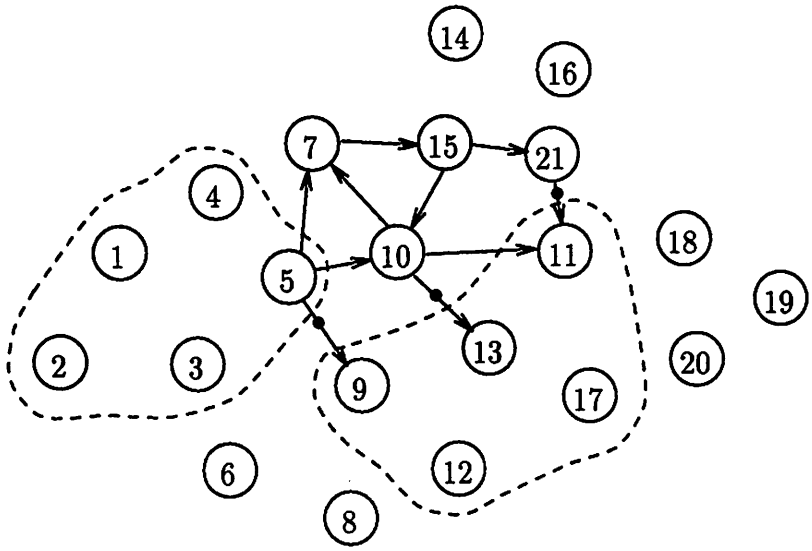

Figure 8: Over-approximation V_2', formed by restricting V_2 to edges on paths from I to F.

$$fromI \;=\; \{5{\rightarrow}10, 15{\rightarrow}10\}.$$

Suppose lines 15 to 18 make the following approximations:

$$\begin{aligned} falseToF &= \emptyset, \\ trueToF &= \{10{\rightarrow}11\}, \\ falseFromI &= \emptyset, \\ trueFromI &= \{15{\rightarrow}10\}. \end{aligned}$$

These two true edges are added to U_3 to yield U_4 (Figure 12).

Iteration 4 Reachability in U_4 adds state 10 to I, and also to F. At this point, I and F intersect, and the algorithm returns "yes, I can reach F."

5 Approximating Boolean functions

Our algorithm to decide state reachability efficiently is predicated upon being able to find close approximations to sets of edges (e.g., the graph $G(x,y)$, and the sets $toF(x,y)$ and $fromI(x,y)$), which have small BDDs. In this section, we define a general problem whose solution can be used to approximate sets of edges. We discuss the related work of Ravi and Somenzi, and then offer our own heuristic to solve the problem.

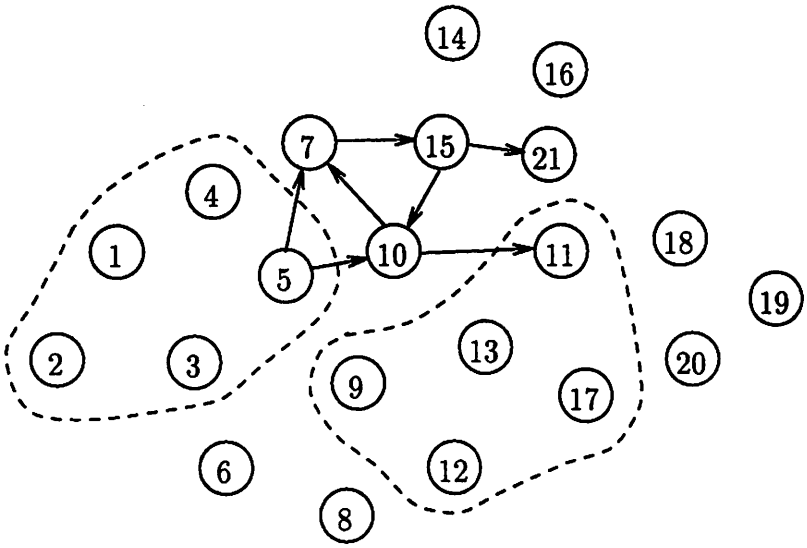

Figure 9: Over-approximation V_3, formed by removing edges 10→13, 21→11 and 5→9 from V_2'.

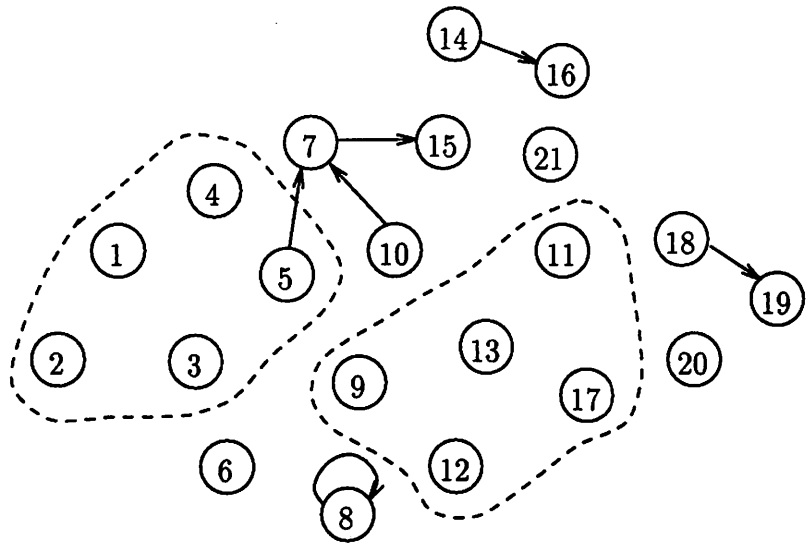

Figure 10: Under-approximation U_3, formed by adding edge 5→7 to U_2.

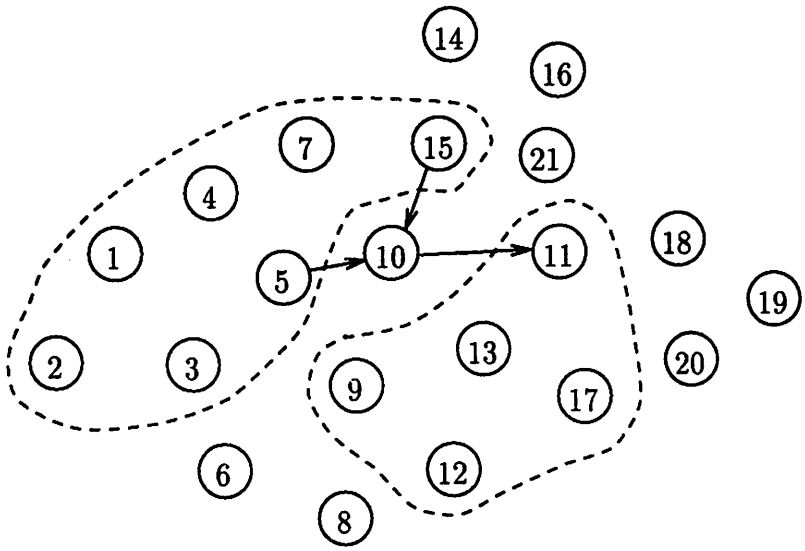

Figure 11: Over-approximation V_3', formed by restricting V_3 to edges on paths from I to F.

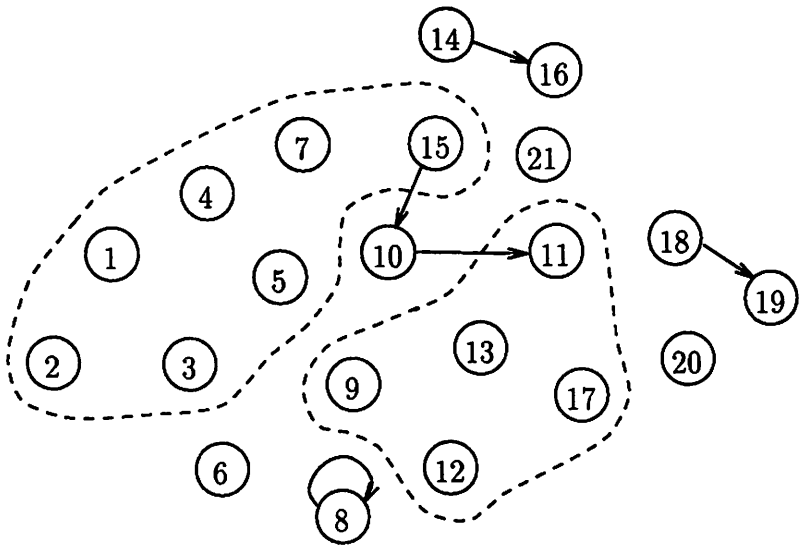

Figure 12: Under-approximation U_4, formed by adding edges 10→11 and 15→10 to U_3.

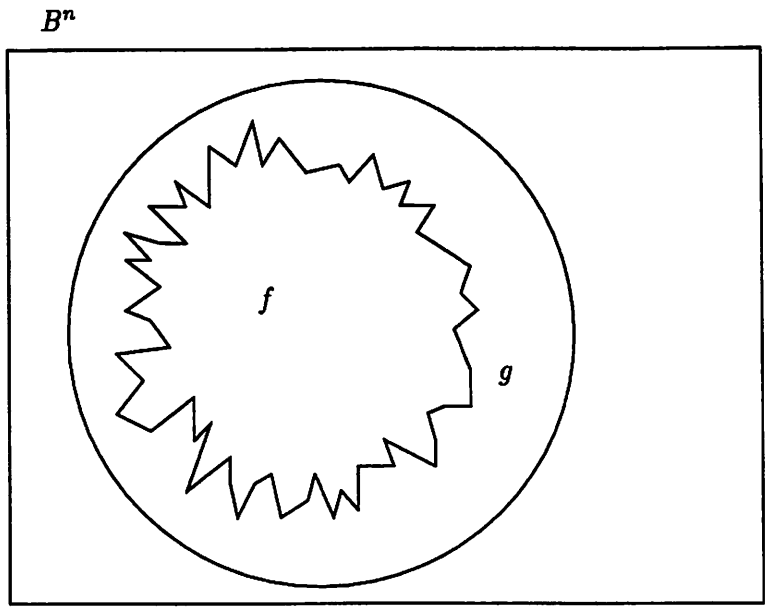

Figure 13: The BDD over-approximation problem.

5.1 Statement of the problem

Given a Boolean function f, we say that the Boolean function g if an *over-approximation* of f is $g \supseteq f$. This definition can be extended to relations by considering the characteristic functions of relations. We want to find an over-approximation g of f such that g just "barely" contains f, and yet the BDD for g is much smaller that the BDD for f (under a fixed variable ordering). This problem is illustrated in Figure 13, where the function f is a "complicated" function with a large BDD, and g is a "simple" function with a small BDD, derived from f by adding some minterms to the onset.

The demands of having a close approximation and yet having a small BDD are sometimes conflicting. There are two extreme approximations we could consider. The first is the function f itself; this approximation is exact, however, by assumption, this function has an unwieldy BDD. The second approximation is the tautology; this approximation has a BDD of size 1, however, it is unlikely to be useful since it does not contain any information. These conflicting demands lead us to the following optimization problem.

Definition 2 *The* BDD over-approximation problem *is, given the BDD for a function* $f : B^n \rightarrow B$ *and* $0 \leq \alpha \leq 1$, *find* $g \supseteq f$ *such that the* cost *of* g *is minimized, where*

$$cost(g) = \alpha(\log_2^2 |onset(g)|) + (1 - \alpha)|BDD(g)|$$

and $|onset(g)|$ *is the size of the onset of* g, *and* $|BDD(g)|$ *is the size of the BDD for* g.

Several remarks regarding this problem are in order.

1. The parameter α appearing in the cost function allows us to control the relative weight between finding a close approximation and finding an approximation with small BDD size. We see that when $\alpha = 1$, the minimum cost solution is f itself, and when $\alpha = 0$, the minimum cost solution is the tautology.

2. Since for an over-approximation g of f, $|\text{onset}(g)| \geq |\text{onset}(f)|$, if $cost(g) < cost(f)$, then this implies that $|\text{BDD}(g)| < |\text{BDD}(f)|$.

3. The logarithm of the onset size is used to balance the two terms being summed. Even though both $|\text{onset}(g)|$ and $|\text{BDD}(g)|$ can be exponential in n, functions we can handle typically have exponential size onsets but polynomial size BDDs. Therefore, so that the onset size term does not dominate the BDD size term, we take the logarithm of the onset size term; but then we square it so that it is not too small.

5.1.1 Complexity of the BDD over-approximation problem

The decision problem corresponding to the BDD over-approximation problem is in NP.
Instance: A function $f : B^n \rightarrow B$ represented by a BDD, $0 \leq \alpha \leq 1$, and $K < |\text{BDD}(f)|$.
Question: Does there exist $g : B^n \rightarrow B$ such that $g \supseteq f$ and $cost(g) \leq K$?

Proposition 3 *The above problem is in NP.*

Proof We must verify in time polynomial in $|\text{BDD}(f)|$ whether or not a guess g is a solution to the problem. If $|\text{BDD}(g)| > |\text{BDD}(f)|$, then we can immediately dismiss g as a potential solution, by the second remark above. Otherwise, we traverse BDD(g) to determine $|\text{onset}(g)|$; this can be done in time $O(|\text{BDD}(g)|)$. Then we compute $cost(g)$, and verify whether or not $cost(g) \leq K$. If so, we verify that $g \supseteq f$; this can be done by checking that $f \cdot \overline{g} = 0$, which can be done in time $O(|\text{BDD}(f)| \cdot |\text{BDD}(g)|)$. ∎

5.1.2 Minterms versus BDD size

The drawing in Figure 13 is meant to suggest that just by adding a "few" minterms to the onset of f, the BDD size can be drastically reduced. Unfortunately, the truth is not so ideal. We now show that adding one minterm to the onset of f cannot reduce the BDD size of f by more than n, where n is the number of variables. Thus, if $|\text{BDD}(f)|$ is exponential in n, then to get an exponential reduction in the BDD size, we must add an exponential number of minterms, which can no longer be construed as a "close" approximation.

Let $x_1, x_2, \ldots, x_n$ span the space B^n. Without loss of generality, assume the BDD variable ordering is $x_1 < x_2 < \ldots < x_n$, with x_1 being the top variable.

Lemma 4 *Let $f : B^n \rightarrow B$ and let $m \in B^n$ be a minterm in the offset of f. Then $|BDD(f+m)| \leq |BDD(f)| + n$. That is, adding a minterm to the onset of a function cannot increase the size of its BDD by more than n.*

Proof We argue that the number of nodes at level i cannot increase by more than 1. Since there are at most n levels, the total size cannot increase by more than n.

Let $b_j \in B$ denote an assignment to x_j, and let m be the minterm $b'_1, \ldots, b'_n$. Consider the cofactors of f on all combinations of $b_1, \ldots, b_{i-1}$. Partitions these cofactors into equivalence classes based on equality. The number of classes whose representative depends on x_i gives the number of nodes at level i in the BDD for f.

Now consider the cofactors of $f + m$ on all combinations of $b_1, \ldots, b_{i-1}$. First, we note that cofactoring distributes over disjunction, so

$$(f + m)_{b_1,\ldots,b_{i-1}} = f_{b_1,\ldots,b_{i-1}} + m_{b_1,\ldots,b_{i-1}}.$$

The cofactor of m by $b_1, \ldots, b_{i-1}$ is 0 for every combination of $b_1, \ldots, b_{i-1}$, except for $b'_1, \ldots, b'_{i-1}$. Thus, all but one of the cofactors of $f + m$ are the same as the corresponding cofactors of f. Hence, each cofactor remains in the same equivalence class, with the exception of the cofactor by $b'_1, \ldots, b'_{i-1}$. In the case that this cofactor forms its own class, and is dependent on x_i, the number of nodes at level i will increase by one. In all other cases (the cofactor joins another class, or forms its own class but is independent of x_i), the number of nodes at level i does not increase. ∎

Lemma 5 *Let $f : B^n \rightarrow B$ and let $m \in B^n$ be a minterm in the onset of f. Then $|BDD(f \cdot \overline{m})| \leq |BDD(f)| + n$. That is, removing a minterm from the onset of a function cannot increase the size of its BDD by more than n.*

Proof The proof is similar to Lemma 4. In this case, we have

$$(f \cdot \overline{m})_{b_1,\ldots,b_{i-1}} = f_{b_1,\ldots,b_{i-1}} \cdot \overline{m}_{b_1,\ldots,b_{i-1}}.$$

The only combination of $b_1, \ldots, b_{i-1}$ where $\overline{m}_{b_1,\ldots,b_{i-1}} \neq 1$ is $b'_1, \ldots, b'_{i-1}$. Thus, all but one of the cofactors of $f \cdot \overline{m}$ are the same as the corresponding cofactors of f. The rest of the proof is the same. ∎

Theorem 6 *Let $f : B^n \rightarrow B$ and let $m \in B^n$ be a minterm. Changing the value of f on m cannot change the size of the BDD for f by more than n.*

Proof

Case 1: m is in the offset of f. By Lemma 4, $|\mathrm{BDD}(f+m)| \leq |\mathrm{BDD}(f)|+n$. We must show that $|\mathrm{BDD}(f+m)| \geq |\mathrm{BDD}(f)|-n$. For sake of contradiction, suppose $|\mathrm{BDD}(f+m)| < |\mathrm{BDD}(f)|-n$. Let $g = f+m$. By Lemma 5, $|\mathrm{BDD}(g \cdot \overline{m})| \leq |\mathrm{BDD}(g)|+n$. Since

$$g \cdot \overline{m} = (f+m)\overline{m} = f \cdot \overline{m} = f,$$

then substituting for g gives $|\mathrm{BDD}(f)| \leq |\mathrm{BDD}(f+m)|+n$. By hypothesis, $|\mathrm{BDD}(f+m)| < |\mathrm{BDD}(f)|-n$, which implies $|\mathrm{BDD}(f)| < |\mathrm{BDD}(f)|-n+n = |\mathrm{BDD}(f)|$, an obvious contradiction. Thus, adding a minterm to f cannot change the BDD size by more than n.

Case 2: m is in the onset of f. By Lemma 5, $|\mathrm{BDD}(f \cdot \overline{m})| \leq |\mathrm{BDD}(f)|+n$. To show that $|\mathrm{BDD}(f \cdot \overline{m})| \geq |\mathrm{BDD}(f)|-n$, we proceed exactly as in Case 1, using Lemma 4 this time. Thus, removing a minterm from f cannot change the BDD size by more than n. ∎

Thus, we see that the effect on BDD size of adding minterms to a function is somewhat gradual. Adding k minterms can reduce the BDD size by at most kn. Of course, adding minterms can also *increase* the BDD size, so choosing which minterms to add requires judiciousness.

5.1.3 Under-approximations

So far we have discussed only the problem of finding good over-approximations. However, we are also interested in finding good under-approximations. The formal statement of the BDD under-approximation problem follows.

Definition 7 *The* BDD under-approximation problem *is, given* $f : B^n \rightarrow B$ *and* $0 \leq \alpha \leq 1$, *find* $g \subseteq f$ *such that the* cost *of* g *is minimized, where*

$$cost(g) = \alpha(\log_2^2(|offset(g)|)) + (1-\alpha)|BDD(g)|.$$

Thus, to minimize the cost, we want to minimize the size of the BDD and the size of the offset, subject to the constraint that $g \subseteq f$. Note that when $\alpha = 1$, the minimum cost solution is f itself, and when $\alpha = 0$, the minimum cost solution is the zero function.

5.2 The subsetting problem of Ravi and Somenzi

Ravi and Somenzi independently and concurrently formulated a problem, termed the *subsetting problem*, which is nearly identical to our BDD under-approximation problem [17]. They employ subsetting to compute an under-approximation of the set of reachable states of an FSM, as explained in Section 3. Here, we explain the subsetting problem, and the heuristics they propose for solving this problem.

Definition 8 *The* subsetting problem *is, given a BDD for $f : B^n \rightarrow B$ and a threshold $K < |BDD(f)|$, find a function g such that $g \subseteq f$, $|BDD(g)| \leq K$, and the number of minterms in the onset of g is maximum.*

This problem is nearly identical to ours, the only difference being that Ravi and Somenzi use the threshold K to control the degree of approximation, whereas we use the parameter α.

The first heuristic they propose is termed *heavy branch subsetting.* This method starts at the root of the BDD for f and follows a single path through the BDD, setting to the constant ZERO the side branches along this path. For a given node on this path, it always sets to ZERO that child "holding" the lesser number of minterms in the onset of f (the "light" child), and keeping the other (the "heavy" child). The procedure terminates when enough nodes have been eliminated so that the total BDD size falls below the threshold K. The procedure keeps track of how many nodes are being eliminated by computing, in a preprocessing step, the number of nodes "held" by each light child, exclusive of its corresponding child (called the *differential_node_count*). Using this technique, the total runtime is linear in $|\text{BDD}(f)|$. The result of this procedure is a BDD with a string of nodes at the top, each with one child pointing to ZERO.

The second heuristic is called *short path subsetting.* The idea here is to keep just those short paths from the root to the constant ONE, because they hold a large number of minterms but cost little in terms of the number of BDD nodes. The procedure first labels each node with the sum of its shortest distance from the root, and its shortest distance to the constant ONE; this is called the *path_length.* Then, based on the threshold K, it determines a maximum value for *path_length* such that removing all nodes with a *path_length* greater than the maximum will yield a BDD of size less that K. The resulting BDD may have many disjoint paths, and consequently little sharing of BDD nodes.

Experiments were conducted to compute under-approximations of the set of reachable states for several large FSMs. These experiments validated the utility of subsetting. As a side note, neither heuristic was shown to be superior to the other.

5.3 Heuristic for the BDD under-approximation problem

For a function f with k minterms in its onset, there are 2^k functions $g \subseteq f$. Since we want to find an approximation g with lower cost than f, g must have fewer BDD nodes. We try to find such a g by replacing some subgraphs of f by the constant ZERO; this is also the general approach of Ravi and Somenzi. This is guaranteed to reduce the number of nodes, while meeting the condition that $g \subseteq f$. However, depending on the value of α, replacing a subgraph by ZERO may actually increase the cost. The challenge is to determine which set of subgraphs to replace by ZERO in order to maximize the cost reduction.

Although we have not been able to determine a lower bound on the complexity of the BDD under-approximation problem, it seems likely that solving the problem exactly would be prohibitive. In fact, since we want to repeatedly apply the approximation operator on BDDs of tens of thousands of nodes, we require an algorithm that is linear, or nearly so. Because of this, we take a very greedy

approach.

The basic idea is to visit the nodes of the BDD for f on a level-by-level basis, from top to bottom. Within a level, the nodes are visited in an arbitrary order. When a node v is visited, we compute

- *numOnset*(v), which is the number of minterms in the onset of f that would be removed if all edges pointing to v were redirected to ZERO, and
- *nodeSavings*(v), which is the number of nodes in the subgraph rooted at v that would be saved if v were replaced by ZERO. Note that some nodes in the subgraph of v are shared by other parts of the BDD, and hence do not contribute to *nodeSavings*(v).

Given these two measures, we can determine whether or not replacing v by ZERO will increase or decrease the overall cost; if it will decrease, then we greedily make the replacement. We continue processing each node in turn until all non-constant nodes have been processed.

We now detail the four major steps of the algorithm, and illustrate it on the BDD in Figure 14. To simplify the presentation, we assume that complement pointers are not used in the BDD. However, the implementation must ultimately take into account complement pointers, because all present-day BDD packages use them. The main complication is that replacing a node v by ZERO will actually *add* minterms to the onset of f, if v can be reached by an odd number of complement pointers.

5.3.1 Step 1: Compute the *onsetFraction* of each node v in f

For the Boolean function rooted at v, *onsetFraction*(v) gives the ratio of the size of the onset to the size of the entire Boolean space. This figure can be computed for all nodes of f in linear time by applying DFS from the root of f. The terminal cases of the recursion are

$$\begin{aligned} \textit{onsetFraction}(\text{ONE}) &= 1, \text{ and} \\ \textit{onsetFraction}(\text{ZERO}) &= 0. \end{aligned}$$

The *onsetFraction* of a non-constant node is computed in terms of the *onsetFraction* of its two children:

$$\textit{onsetFraction}(v) = \frac{1}{2}\textit{onsetFraction}(v.\textit{left}) + \frac{1}{2}\textit{onsetFraction}(v.\textit{right}).$$

The *onsetFraction* for each node in our example BDD is shown in Table 1. For example,

$$\textit{onsetFraction}(C) = \frac{1}{2}\cdot\frac{3}{8} + \frac{1}{2}\cdot\frac{7}{8} = \frac{5}{8}.$$

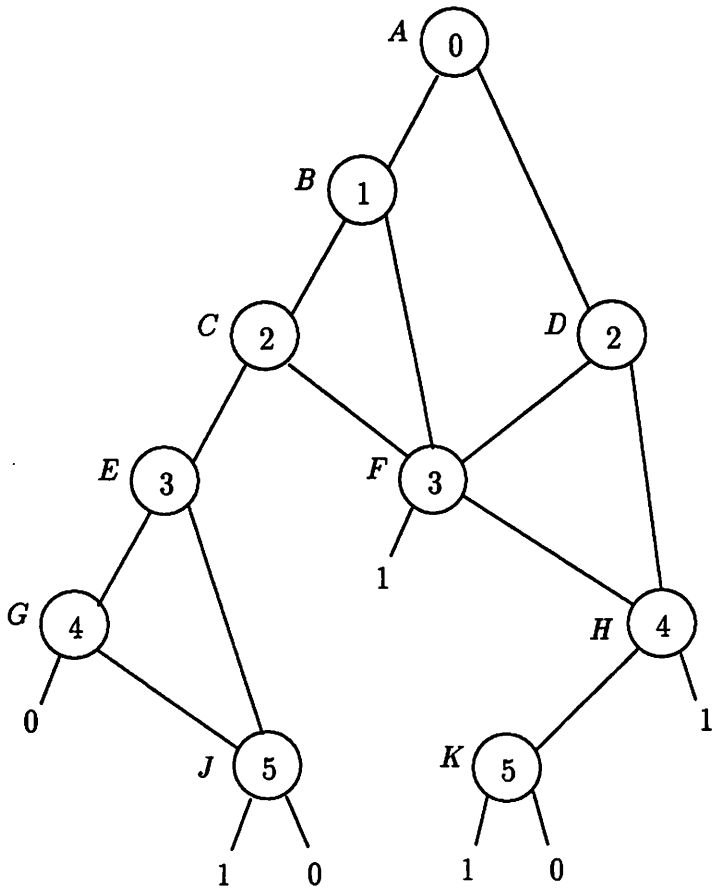

Figure 14: BDD used to illustrate the *bddUnderApprox* algorithm.

node	*onset-Fraction*	*function-RefCount*	*node-Savings*	*num-Minterms*	*num-Onset*	*costBenefit*, α					
						0	.2	.4	.6	.8	1
A	25/32	1	10	64	50	10	3.7	-2.6	-8.9	-15.2	-21.5
B	3/4	1	5	32	24	5	1.4	-2.2	-5.8	-9.4	-13.0
C	5/8	1	4	16	10	4	1.9	-0.2	-2.3	-4.4	-6.5
D	13/16	1	1	32	26	1	-2.0	-4.9	-7.9	-10.8	-13.8
E	3/8	1	3	8	3	3	2.0	0.9	-0.1	-1.2	-2.2
F	7/8	3	1	40	35	1	-2.6	-6.2	-9.8	-13.4	-17.0
G	1/4	1	1	4	1	1	0.6	0.3	-0.1	-0.4	-0.8
H	3/4	2	2	36	27	2	-1.2	-4.5	-7.7	-11.0	-14.2
J	1/2	2	1	6	3	1	0.4	-0.3	-0.9	-1.6	-2.2
K	1/2	1	1	18	9	1	-0.4	-1.8	-3.2	-4.6	-6.0

Table 1: The *bddUnderApprox* algorithm applied to the BDD of Figure 14.

5.3.2 Step 2: Compute the *functionRefCount* of each node v in f

functionRefCount(v) gives the number of edges pointing to v from within the function f; it excludes pointers from other functions within the same BDD manager. This figure can be computed for all nodes of f in linear time by performing BFS from the root. The *functionRefCount* of each node is initialized to 0. Then, for each node visited, the *functionRefCount* of each of its children is incremented by one. The *functionRefCount* of each node of our example BDD is shown in Table 1.

5.3.3 Step 3: Approximate the BDD

This step is the heart of the procedure. The nodes are visited via BFS. The subgraph rooted at a node is replaced by ZERO if this reduces the overall cost of the solution. When this happens, the *functionRefCounts* of some nodes in the subgraph are decremented.

The details of this step are now given. For each node v visited that has a non-zero *functionRefCount*, the following three actions are performed.

Action 1: Compute *nodeSavings*(v), the number of nodes that would be eliminated in f if just the subgraph rooted at v was replaced by ZERO. This can be computed by performing a local BFS starting from v. Each node has a *localRefCount*, which is initialized to *functionRefCount* each time a local BFS is commenced. When a node u is visited during a BFS, if its *localRefCount* is non-zero, then u is not explored further, and it does not contribute to *nodeSavings* of v; a non-zero *localRefCount* indicates that such a node is being shared by other parts of the BDD for f. On the other hand, if the *localRefCount* of u is zero, then *nodeSavings*(v) is incremented, and the *localRefCounts* of u's two children are decremented by one.

Consider computing *nodeSavings* for node B in our example. By definition, B itself contributes 1 to *nodeSavings*(B). The *localRefCount* of the children of B are decremented: for C, 1 is decremented to 0, and for F, 3 is decremented to 2. Next, C is visited, and since its *localRefCount* is now 0, *nodeSavings*(B) is incremented (to 2), and the *localRefCounts* of E and F are decremented, to 0 and 1, respectively. Say F is visited next. Its *localRefCount* is not 0, so we skip over F and proceed to E. Its *localRefCount* is 0, and proceeding in this fashion, we see that E, G and F all contribute to *nodeSavings*(B). Hence, *nodeSavings*(B) is 5. The *nodeSavings* for other nodes is shown in Table 1.

This step, repeated for each node, can lead to overall quadratic running time (consider a BDD that is just a single chain of nodes; BFS from each node will explore the rest of the chain). However, because the local BFS search from a node is pruned at nodes whose *localRefCount* is non-zero, the running time in practice should be nearly linear.

Action 2: Compute *numOnset*(v), the number of minterms in the onset of f that would be removed if v was replaced by ZERO. This can be computed by multiplying *onsetFraction*(v) by *numMinterms*(v). *NumMinterms*(v) records how many of the 2^n minterms of the Boolean space "pass through" v. For the root of f, *numMinterms* is 2^n. As each node u (that is not replaced by

ZERO) is visited in the global BFS of Step 3, *numMinterms* of each child of u is incremented by one-half of *numMinterms*(u).

In our example,

$$\begin{aligned} \mathit{numMinterms}(A) &= 2^6 = 64, \\ \mathit{numMinterms}(B) &= \frac{1}{2}\mathit{numMinterms}(A) = 32, \\ \mathit{numMinterms}(F) &= \frac{1}{2}\mathit{numMinterms}(B) + \frac{1}{2}\mathit{numMinterms}(C) + \frac{1}{2}\mathit{numMinterms}(D) \\ &= 16 + 8 + 16 = 40 \end{aligned}$$

Hence,

$$\begin{aligned} \mathit{numOnset}(F) &= \mathit{onsetFraction}(F) \cdot \mathit{numMinterms}(F) \\ &= \frac{7}{8} \cdot 40 = 35. \end{aligned}$$

For each node, *numOnset* can be computed in constant time.

Action 3: Compute *costBenefit*(v), which measures the change in cost of the solution if v were replaced by ZERO, to yield the function f_{new}. This is computed as follows.

$$\begin{aligned} \mathit{costBenefit} &= \mathit{cost}(f) - \mathit{cost}(f_{new}) \\ &= [\alpha(\log_2^2 |\mathit{offset}(f)|) + (1-\alpha)|\mathrm{BDD}(f)|] \\ &\quad -[\alpha(\log_2^2 |\mathit{offset}(f_{new})|) + (1-\alpha)|\mathrm{BDD}(f_{new})|] \\ &= \alpha[\log_2^2 |\mathit{offset}(f)| - \log_2^2 |\mathit{offset}(f_{new})|] \\ &\quad +(1-\alpha)(|\mathrm{BDD}(f)| - |\mathrm{BDD}(f_{new})|) \\ &= \alpha[\log_2^2(|\mathit{offset}(f)|) - \log_2^2(|\mathit{offset}(f)| + \mathit{numOnset}(v))] \\ &\quad +(1-\alpha)\mathit{nodeSavings}(v) \end{aligned}$$

If *costBenefit*(v) is greater than zero, then the flag *replaceByZero*(v) is set, and the *functionRefCount* of v's two children are decremented by one. If this causes *functionRefCount* of a child to fall to zero, then the *functionRefCounts* are recursively decremented. For example, if *costBenefit*(B) is greater than zero, *functionRefCount*(F) will fall to one, and *functionRefCount* of C, E and G will fall to zero.

If *costBenefit*(v) is less than or equal to zero, then the flag *replaceByZero*(v) is reset, and *numMinterms* of each child of v is incremented by one-half of *numMinterms*(v).

The *costBenefit* computed for a node v is affected by which other nodes have been marked for replacement by ZERO. In particular, *numMinterms*(v) may decrease, and *nodeSavings*(v) may increase, as nodes at or above the level of v are marked for replacement by ZERO (of course, if *functionRefCount*(v) falls to zero, then *costBenefit*(v) is irrelevant). By processing the nodes in a top-down fashion, *costBenefit*(v) needs to be computed just once, when v is considered for replacement.

Even though *costBenefit*(v) is affected by the actions above v, the values for *costBenefit* in Table 1 are computed, for illustration purposes only, assuming that no nodes processed before a given node are marked for replacement. Also, since *costBenefit* is a function of α, the value of *costBenefit* is shown for 6 different values of α. As expected, as α tends to one, the *costBenefit* becomes negative, meaning replacement by ZERO is undesirable.

5.3.4 Step 4: Build the new BDD

This process starts from the root and proceeds recursively in DFS fashion. If the constants ZERO or ONE are reached, then that constant is returned. If a node marked *replaceByZero* is reached, then ZERO is returned. Otherwise, for a node labeled by variable x, a new node is created labeled with x and with children formed by the recursive building process.

For our example, suppose $\alpha = 0.4$. Then the first node processed with positive *costBenefit* is E. In fact, this is the only node replaced by ZERO (G becomes irrelevant once E is replaced). The new BDD is shown in Figure 15. Whereas the original BDD had 50 onset minterms and 10 nodes, the new BDD has 47 onset minterms and 7 nodes.

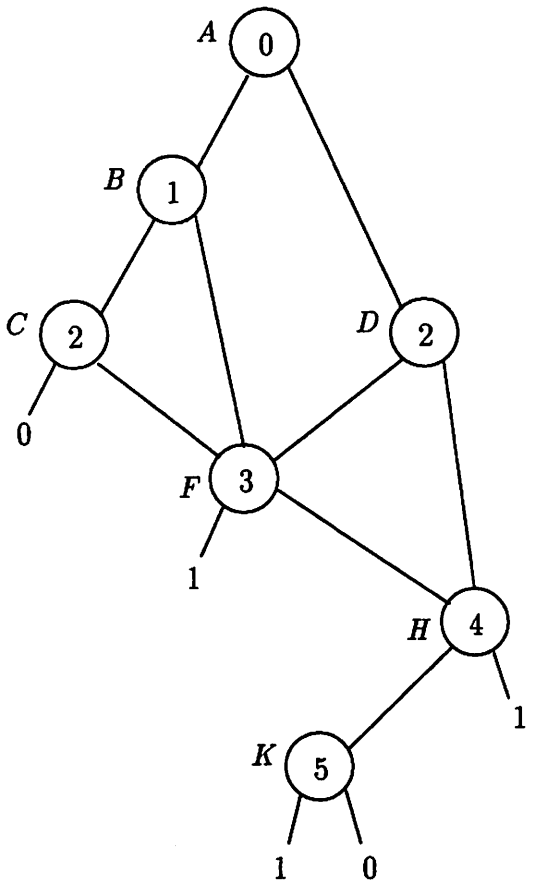

Figure 15: The result of *bddUnderApprox* applied to the BDD of Figure 14. E is replaced by ZERO.

5.3.5 Discussion

Greedily choosing one node at a time for replacement by ZERO may lead to a suboptimal solution. Consider the partial BDD shown in Figure 16. Both of the children of node A are shared by other parts of the function, so *nodeSavings*(A) is one (node A itself). Hence, unless α is nearly zero, A probably will not be replaced by ZERO. Likewise, *nodeSavings*(B) is one and B probably will not be replaced. However, if we considered replacing A and B simultaneously by ZERO, we would find that *nodeSavings*($\{A, B\}$) is $K + 2$, where K is the number of nodes in the common subgraph of A and B. If K is large, this may trigger replacement.

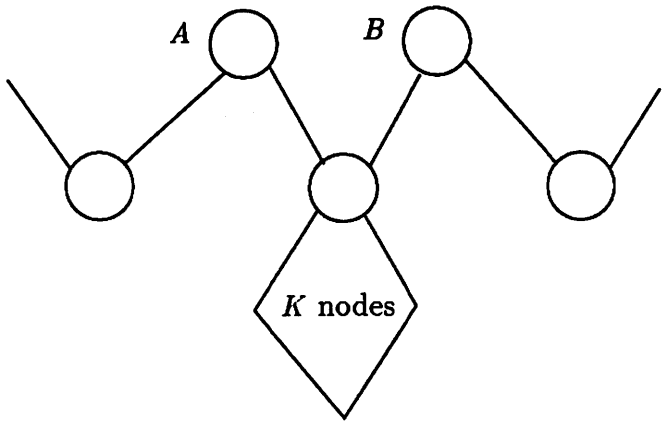

Figure 16: Neither A nor B will be replaced by ZERO when considered individually, but may be replaced by ZERO if considered simultaneously.

The algorithm could be modified easily to consider pairs of nodes for replacement. However, this would increase the complexity of the algorithm, which may make it impractical for intermediate to large BDDs. Also, instead of pairs of nodes, we might want to consider larger sets for replacement.

Another limitation with our approach comes in selecting the value for α. In our example, for $\alpha = 0$, all nodes qualify for replacement, whereas at $\alpha = 0.6$, none of the nodes qualify. Combining the onset size and the BDD size terms in the same equation makes it difficult to select a precise value of α that distinguishes "good" replacements from "bad" replacements. Possibly a threshold-based approach, like that of Ravi and Somenzi, might be more robust.

5.4 Application to binary Boolean operations

In our algorithm for deciding reachability, we frequently want to find a good approximation to the Boolean combination of a pair of functions, for example, the conjunction $f \cdot g$. We *could* define a new BDD operator that takes as input two functions and returns an approximation to their conjunction. Instead, we take an alternate approach where we form the conjunction exactly, and then approximate the result. This approach begs the question if we are able to form the conjunction exactly. We can, as long as we keep small the intermediate BDDs of the reachability computation. In other words, if we apply *bddApprox* to all intermediate BDDs, then we should be able to perform

local computations exactly. A benefit of this approach is that we can concentrate on developing heuristics for just a single problem, the BDD approximation problem.

6 Approximating sets of edges

We concluded Section 4 by listing the three major subroutines of our algorithm to decide reachability. Each of these subroutines involves approximating a set of edges; we now discuss each in detail.

6.1 Initial over-approximation of G

The goal of this subroutine is to find a superset V of the edges of G such that V has low *cost*, as defined in Definition 2. As a reminder, the function we wish to approximate is

$$G(x,y) = \exists u \prod_{i=1}^{l} T_i(x,u,y_i).$$

The first step is to build each T_i. Then, some u variables are "cut" to partition the T_i's into a set of clusters. Next, each cluster is built separately, and finally the clusters are conjuncted to yield the over-approximation V. At each step of this process, *bddOverApprox* is used to control the size of the BDDs.

6.1.1 Building each T_i

As stated earlier, $T_i = (y_i \equiv \delta_i(x,u))$, where δ_i is the next state function of the ith flip-flop. The BDD for δ_i may be too large to build. In this case, T_i can be represented by the conjunction of a set of smaller terms by introducing intermediate variables.[3]

Specifically, starting at the combinational inputs x and u and proceeding in topological order through the combinational network, we begin by constructing the BDD for each network node in terms of the combinational inputs. However, if the BDD for the function g_j of a given node v_j exceeds a user-settable threshold, then an intermediate variable p_j is introduced at node v_j. Then the BDDs of nodes in the fanout of v_j are built in terms of p_j. T_i can then be expressed as the conjunction of the terms $(p_j \equiv g_j)$, with the intermediate variables existentially quantified. Each of these terms can be over-approximated using *bddOverApprox* so that their product has a reasonable BDD size. In the sequel, we refer to the result of this step as $T_i(x,u,y_i)$, regardless of whether or not T_i has in fact been approximated.

[3]This technique has been used by others in a variety of settings, e.g., [14, 13, 15].

6.1.2 Cutting u variables

The next step is to "cut" some u variables to partition the T_i's into a set of clusters. This approximation relies on the observation that $(\exists x f) \cdot (\exists x g) \supseteq \exists x (f \cdot g)$. The idea is to cut some of the u variables by moving them into the product. For example, we might cut u_1 by replacing

$$\exists u_1, u_2 \, [T_1(x, u_1, u_2, y_1) \cdot T_2(x, u_1, y_2)]$$

by

$$(\exists u_1, u_2 \, T_1(x, u_1, u_2, y_1)) \cdot (\exists u_1 \, T_2(x, u_1, y_2)).$$

Equivalently, the problem is to cluster the T_i's; any u variables passing between clusters are cut. We want to minimize the number of cut variables, so that the amount of over-approximation is minimized. We formulate the problem as a traditional graph partitioning problem on hypernets. In particular, we create an undirected graph, where each flip-flop is represented by a vertex, and there exists an edge labeled by u_k between vertices i and j if T_i and T_j both depend on u_k. Then we successively apply graph bipartitioning (using, for example, the Fiduccia-Mattheyses algorithm), minimizing the number of u variables cut. The size of each partition is limited by a user provided parameter, giving the maximum of the sum of BDD sizes for each partition.[4]

6.1.3 Building each cluster

At this point, we must construct the graph $C_j(x, y)$ for each cluster:

$$C_j(x, y) = \exists u \prod_{i \in J} T_i(x, u, y_i)$$

where J is the set of flip-flops in the jth partition. First, we find a schedule for the conjunctions and quantifications (for example, using the techniques in [12]). In general, this may be in the form of a tree. Then we build C_j according to this schedule, but we apply *bddOverApprox* to intermediate results to avoid large BDDs. In particular, there are two types of intermediate computations.

1. Conjunction: form the conjunction exactly and then apply *bddOverApprox* to the result.
2. Existential quantification: apply *bddOverApprox* to the results of the intermediate disjunctions (i.e., $f_x + f_{\overline{x}}$), and to the final result of existential quantification.

6.1.4 Conjuncting the clusters

The last step is to take the product of the clusters C_j. Again, we form each conjunction exactly, and then apply *bddOverApprox* to the result. The final result is the approximation V.

V should have no more than approximately 10,000 BDD nodes, so that we can manipulate it efficiently. Hence, we need some dynamic control to make sure that V does not exceed this limit.

[4] The flip-flop partitioning technique of Cho *et al.* [7] could also be applied to the present problem.

This could take the form of stopping the computation when the limit is exceeded, and restarting it with a lower value of α (i.e., a worse approximation, but smaller BDD size); or we could just restart the phase of conjuncting clusters with a lower value of α.

6.2 Initial under-approximation of G

The goal of this subroutine is to find a subset U of the edges of G such that U has low *cost*, as defined in Definition 7. Computing U follows the same outline as computing the over-approximation V.

We cut the u variables using the same partition found in computing V. However, rather than *existentially* quantifying the cut variables, we now *universally* quantify them, relying on the fact that $(\forall x f) \cdot (\forall x g) \subseteq \exists x (f \cdot g)$. To build each cluster, we use under-approximation on intermediate results, rather than over-approximation.

6.3 Approximation of edges from I in V

The variable *fromI* in the reachability algorithm contains those edges in the current over-approximation V that pass from a state in I to a state not in I. The set *fromI* can be partitioned into two sets.

1. True edges: these are edges that exist in the exact graph G and that, when added to the under-approximation U (line 28), allow the forward traversal in U to progress (line 6).
2. False edges: these are edges that do *not* exist in G, and that, when removed from V (line 27), further restrict the set of potential paths from I to F (line 10).

The set of true edges E is the set of *all* edges from I in V, restricted to the exact graph $G(x, y)$:

$$\begin{aligned} E(x,y) &= G(x,y) \cdot \mathit{fromI}(x,y) \\ &= (\exists u \prod_{i=1}^{l} T_i(x,u,y_i)) \cdot \mathit{fromI}(x,y) \end{aligned}$$

The set of false edges is then just the set difference of E from V, $V(x,y) \setminus E(x,y)$.

Ideally, we would like to determine the partition exactly. Unfortunately, this problem is hard, as deciding if just a single edge is true or false is already NP-complete. Thus, we settle for approximating the sets of true and false edges. We want to find some edges from I that are definitely false (line 17), and some that are definitely true (line 18); the status of the remainder of the edges in *fromI* will be unknown.

Clearly, an under-approximation of E yields an approximation to the true edges, and an over-approximation of E yields an approximation to the false edges. To compute these approximations of E, we would like to rewrite the equation for E so that it has the same form as the equation for

G, thus permitting the application of the procedures outlined in Sections 6.1 and 6.2. This can be done simply:

$$\begin{aligned} E(x,y) &= \exists u[(\prod_{i=1}^{l} T_i(x,u,y_i)) \cdot \mathit{fromI}(x,y)] \\ &= \exists u[\prod_{i=1}^{l} (T_i(x,u,y_i) \cdot \mathit{fromI}(x,y))] \\ &= \exists u \prod_{i=1}^{l} T_i'(x,u,y_i) \end{aligned}$$

where $T_i' = T_i \cdot \mathit{fromI}$.[5] Thus, we have expressed the set E of true edges from I in V as the product of individual transition relations, and we can apply the procedures of Sections 6.1 and 6.2 to compute an over-approximation and under-approximation, respectively. Figure 17 illustrates the various sets involved in the above computation.

$$\begin{aligned} G &= \text{exact graph} \\ V &= \text{current over-approximation of } G \\ U &= \text{current under-approximation of } G \\ \mathit{fromI} &= \text{edges from } I \text{ in } V \\ E &= G \cap \mathit{fromI} \\ V' &= \text{over-approximation of } E \\ U' &= \text{under-approximation of } E \\ \text{approx. false edges} &= \mathit{fromI} \setminus V' \\ \text{approx. true edges} &= U' \end{aligned}$$

Note that V does not necessarily contain G, because some true edges are removed from V at line 10.

The procedure for approximating the true and false edges to F follows analogously.

7 Summary and future work

We have presented a technique for deciding state reachability for large FSMs. Specifically, we seek to answer if there exists a path from a set I of initial states to a set of F of final states in an FSM. Several problems in logic synthesis, formal verification, and testing can be reduced to this question, and hence an efficient algorithm for solving this problem would have great benefit.

Our approach constructs an over-approximation V, and an under-approximation U, to the state transition graph G. Then, the potential witness paths from I to F in V are used to guide the search for a true path in U from I to F.

[5] Alternatively, we could define $T_i' = \mathit{restrict}(T_i, \mathit{fromI})$, using the *restrict* operator of [8] . Then $\exists u \prod T_i'$ is no longer exactly $E(x,y)$, but can still be used to form approximations.

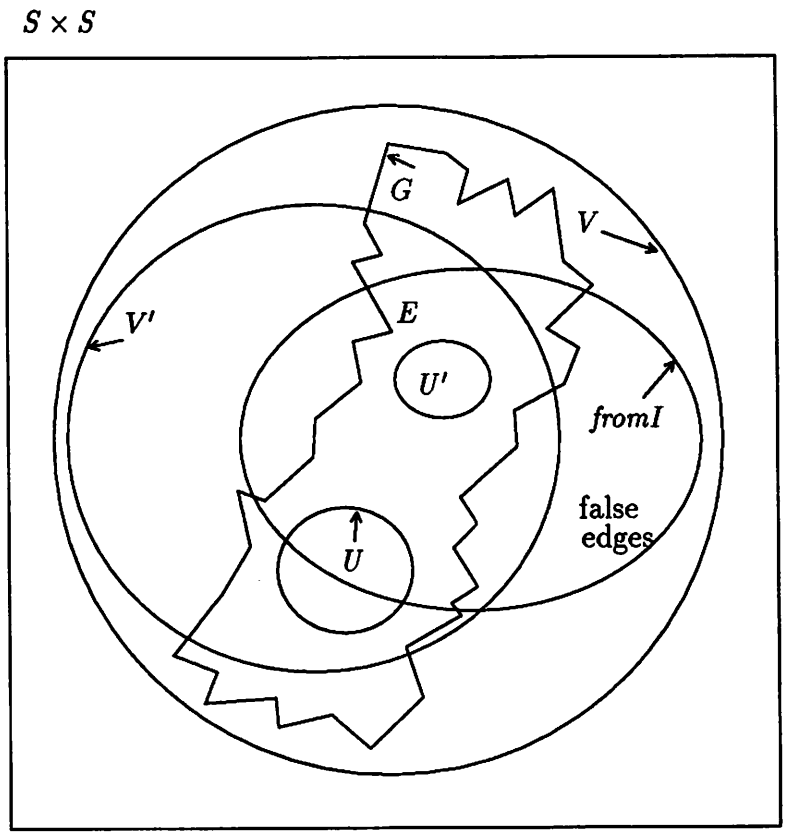

Figure 17: Diagram showing over- and under-approximations to $E(x, y)$.

The success of our approach hinges on the quality of the approximations that we construct. A good approximation is one that retains most of the original information, yet has a small representation. We use BDDs to represent the set of edges of a state graph. We have formulated a general optimization problem, called the *bddApprox* problem, which seeks to find a set representing a close approximation of another set, and yet having a small BDD representation. We presented a heuristic for solving the *bddApprox* problem. Ravi and Somenzi formulated the same problem, and presented several heuristics. We suspect that others will find applications for the *bddApprox* problem, and will develop more heuristics for its solution.

As with any heuristic for solving a hard problem, our approach can ultimately be validated only by implementing the algorithms and testing them on a set of examples. This remains as future work.

References

[1] Adnan Aziz and Robert K. Brayton. Verifying interacting finite state machines. Technical Report UCB/ERL M93/52, Electronics Research Laboratory, U.C. Berkeley, July 1993.

[2] Felice Balarin. *Iterative Methods for Formal Verification of Digital Systems.* PhD thesis, University of California, Berkeley, 1994.

[3] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. 28th Design Automat. Conf.*, pages 403–407, June 1991.

[4] Gianpiero Cabodi and Paolo E. Camurati. Exploiting cofactoring for efficient FSM symbolic traversal based on the transition relation. In *Proc. Int'l Conf. on Computer Design*, pages 299–303, October 1993.

[5] Gianpiero Cabodi, Paolo E. Camurati, and Stefano Quer. Efficient state space pruning in symbolic backward traversal. In *Proc. Int'l Conf. on Computer Design*, pages 230–235, October 1994.

[6] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Bernard Plessier, and Fabio Somenzi. Algorithms for approximate FSM traversal. In *Proc. 30th Design Automat. Conf.*, pages 25–30, June 1993.

[7] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Massimo Poncino, and Fabio Somenzi. A structural approach to state space decomposition for approximate reachability analysis. In *Proc. Int'l Conf. on Computer Design*, pages 236–239, September 1994.

[8] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, November 1989.

[9] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[10] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, October 1992.

[11] Daniel Geist and Ilan Beer. Efficient model checking by automated ordering of transition relation partitions. In David L. Dill, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310, Stanford, CA, June 1994. Springer-Verlag.

[12] Ramin Hojati, Sriram C. Krishnan, and Robert K. Brayton. Early quantification and partitioned transition relations. In *Proc. Int'l Conf. on Computer Design*, October 1996.

[13] Jawahar Jain, Amit Narayan, Claudionor Coelho, Sunil P. Khatri, Alberto L. Sangiovanni-Vincentelli, Robert K. Brayton, and Masahiro Fujita. Combining Top-down and Bottom-up Approaches for ROBDD Construction. Technical Report UCB/ERL M95/30, Electronics Research Laboratory, U.C. Berkeley, April 1995.

[14] Patrick C. McGeer, Kenneth L. McMillan, Alexander Saldanha, Alberto L. Sangiovanni-Vincentelli, and Patrick Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 402–407, November 1995.

[15] Rajeev K. Ranjan. Private communication, 1996.

[16] Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Bernard Plessier, and Carl Pixley. Efficient BDD algorithms for FSM synthesis and verification. In *International Workshop on Logic Synthesis*, pages 3–27 – 3–34, May 1995.

[17] Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 154–158, November 1995.

[18] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. 31st Design Automat. Conf.*, pages 225–231, San Diego, CA, June 1994.

[19] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 130–133, November 1990.