Incremental Methods for FSM Traversal

Gitanjali M. Swamy Vigyan Singhal Robert K. Brayton
Department of Electrical Engineering and Computer Science.
University of California at Berkeley
Berkeley, CA 94720

Abstract

Computing the set of *reachable states* of a finite state machine is an important component of many problems in the synthesis, and formal verification of digital systems. Computing the reachable states is computationally expensive due to the "explosion" in the number of states in real designs. However, the process of design is usually iterative, and the designer may modify and recompute information many times. Unfortunately, the reachability computation is called each time the designer modifies the system, because current methods for reachability analysis are not incremental.

The current method for representation of the reachable states is inherently non-updatable. We solve this problem by presenting alternate ways to represent the reachable set, and incremental algorithms that can update the new representation each time the designer changes the system. The incremental algorithms use the reachable set computed at a previous iteration and information about the changes to the system to update it, rather than compute the reachable set from the beginning. Our experimental results demonstrate that this may result in considerable savings in time.

1 Introduction

Reachability is an essential computation in both formal verification [1] as well as sequential synthesis [2] and [3]. 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 *finite state machine* (or FSM) can be represented by a directed graph, which is also called a *state transition graph*. Computing the reachable states of this graph is an essential computation in sequential synthesis and formal verification, and can be done by a breadth first search (BFS) exploration of the state transition graph beginning at the initial states.

Unfortunately this computation explodes when the number of states in the finite state machine becomes very large. This is often called the *state explosion problem*. To overcome this problem, an implicit representation called binary decision diagrams or BDDs [4], is sometimes used to implicitly represent needed relations and sets; e.g. the transition relation of the FSM, the set of reachable states or the set of initial states. When BDDs are used, the steps in the BFS traversal of the state transition graph, can

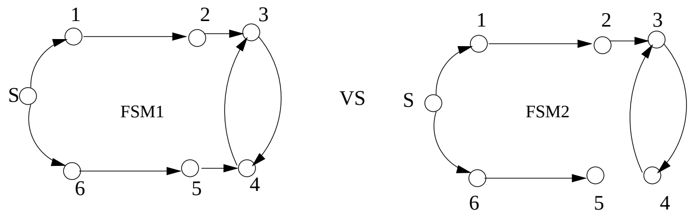

Figure 1: Reachability is non-incremental

be written as fixed point computations of propositional formulae on the transition relation, initial states ([5],[6]).

The process of design is incremental, and the designer may modify the design many times. The current techniques for reachability require that each time the designer modifies the design, the set of reachable states must be re-computed from the beginning. This results in unnecessary re-computation, which is particularly cumbersome in light of the state explosion problem. Instead, it is preferable if the set of reachable states can be updated incrementally at each iteration of the design process.

This paper deals with the construction of such incremental algorithms for reachability. The complete reachability analysis is executed only once, and all successive changes are propagated from the previous iteration. We note that knowing only the set of reachable states obtained from the reachability computation is not sufficient for updating the reachable set. This can be understood by considering the two examples in figure 1. Both have identical reachable states, but if edge $(2,3)$ is deleted then FSM_1 has a different set of reachable states from FSM_2. This is due to the presence or absence of edge $(5,4)$, and if no traversal information is stored then this cannot be determined without examining the entire state space of the two FSM's.

We overcome this problem by storing, rather than a reachable set, either the spanning tree or an acyclic spanning graph, which can be generated during reachability computations. Our algorithm updates the spanning tree, or graph that can be generated by any BFS type procedure for reachability. For example, refer to the FSM in figure 2. Usual reachability algorithms just store 0-1 reachable information, i.e. whether or not the state is reachable. However we store the spanning tree shown, which is effectively computed during reachability. Our algorithm uses information about the changes made to the system, and the original spanning graph of the reachable states to compute a new spanning graph, which represents the new reachable states. An additional bonus of using this way of storing the reachable states is that the BDD variable ordering [4] for a finite state machine is often chosen according to the transition relation. This may not be a good ordering for the reachable states. However, our reachable state representation is much closer to the transition relation, and hence it seems likely that the BDD for it will be compact with the same ordering.

The paper is organized as follows: the basic definitions are given in section 2. Previous work is described in section 3. Section 4 describes how changes to the system are characterized, and two incremental

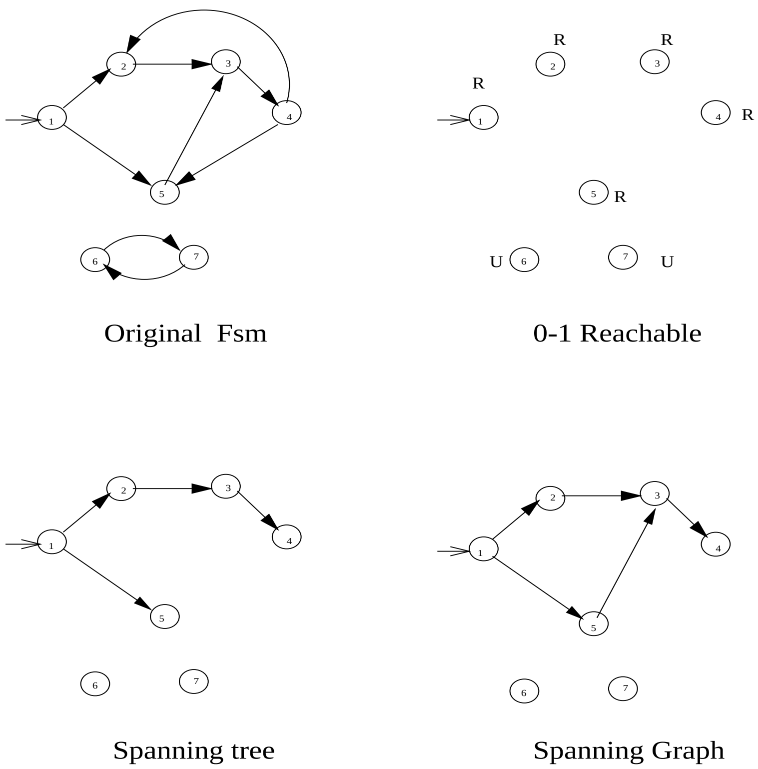

Figure 2: Different ways of storing the reachable states

algorithms for reachability; how this information can be computed, and updated in an implicit framework using BDDs. We present some results in section 5. Finally, we conclude our paper by presenting our conclusions, and future directions for this work.

For brevity, details of proofs of theorems have been omitted; they may be found in [7].

2 Definitions

Definition 1 Finite State Machine: *A finite state machine or* finite automaton $\mathcal{M}$ *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 $(\Sigma \times \Gamma)$.*

Since, we are only concerned with reachable behavior in this paper, we will be using the transition relation after removing input and output dependencies. This will be referred to as $T(x, y)$, where x and y are present state and next state variables respectively. Note that $T(x, y) = \exists_{(\sigma,\gamma)} T(x, y, \sigma, \gamma)$.

Path: *A sequence of states, $r = r_o \ldots r_i \ldots, r_j \in Q$, is a path, or run of T for a word $\sigma = (\sigma_0 \ldots \sigma_i \ldots)$, $\sigma \in \Sigma^\omega$, if $r_0 \in I$ and for $i \geq 0$, $\exists_{\gamma_i} T(r_i, \sigma_i, \gamma_i, r_{i+1}) = 1$.*

Definition 2 Product Machine: *Given a collection of interconnected finite state machines $\{M_1, M_2, \ldots, M_n\}$, their* product *is the finite state machine on the product state space. The transition relation is the cartesian product of the component transition relations.*

Definition 3 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.*

2.1 Some Important Computations

This section describes the computation procedure for some important operators required in this paper. 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.

1. **Fixed point** Let $f(x)$ be a monotone (increasing or decreasing) function, the fixedpoint FP of f given I is given by the set $Q = f^i(I)$, where $f(Q) = Q$ (refer to [8]). If f is monotonically increasing, then the fixed point is called the least fixed point or LFP, and if f is monotonically decreasing, it is called the greatest fixed point or GFP.

   $FP(f(), I)$
     $R = f(I)$
   if $(R = I)$
     return R
   else
     return $FP(f(), R)$

   For example the set of reachable states is computed as the $LFP(f(), I)$ of $f(Q(x)) = Q(x) + \exists_y T(y, x) \cdot Q(y)$, given $I(x)$ the initial states.

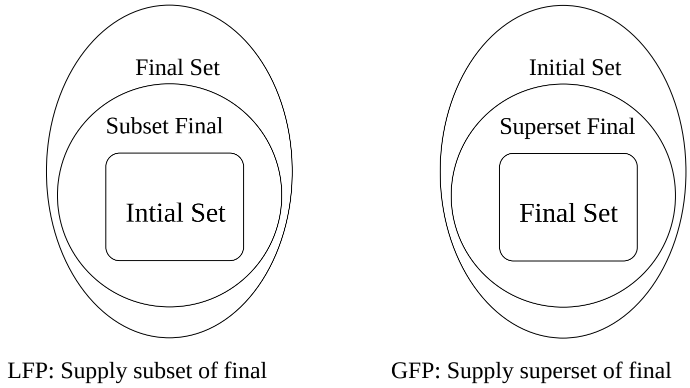

Figure 3: Fixed Points

Our incremental algorithms will use the following fact about fixed points: any subset of the final answer that contains the initial set, returns the correct final answer to an LFP computation, when supplied to it. Figure 3 illustrates this point. A similar statement can be made about GFP computations, as shown by Swamy and Brayton [9].

3 Previous work

Computing the set of reachable states in the transition relation of a finite state machine is equivalent to doing 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] , Coudert and Madre [6], and McMillan [10] extended this concept to handle reachability in larger systems, by using implicit methods. All quantities (transition relations, sets of states etc.) are represented by BDDs (binary decision diagrams), and the algorithm is represented by a fixed point computation (refer to section 2).

Algorithm 3($T(x,y), I(x)$**)**
  $f(Q(x)) = \exists_y T(y,x)Q(y) + Q(x)$
  $R(x) = LFP(f, I(x))$
return $R(x)$

Unfortunately, this algorithm is not incremental, and if the designer modifies the system, the reachable states have to be computed from the beginning. Incremental algorithms for certain graph problems were examined by Ramalingam and Reps [11]. They defined incremental algorithms as algorithms, whose complexity was a function of the size of the change to the system alone. If $\Delta = \Delta_{input} + \Delta_{output}$, and Δ_{input} is the change in the input to the program, Δ_{output} is the change that results in the output, then the complexity is $O(f(\Delta))$. This criterion for incrementality is too stringent for our purposes. In Swamy and Brayton [9], incremental algorithms for methods of formal verification are described. Although reachability is an essential part of any formal verification procedure, the issue of how the reachability computation might be made incremental is not discussed; we intend to address this issue here. We simply require that the incremental algorithm use the answer of a previous reachability iteration to compute a new answer.

4 Incremental Algorithms for Reachability

Let $R(x)$ is a set of reachable states in the system. We want to use information about the changes to the system to incrementally modify $R(x)$. The potential for speedup is that $R(x)$ need not be recomputed from the beginning; intermediate results can be used to avoid unnecessary computations.

Unfortunately, as shown in Figure 1 in section 1, just the old set of reachable states is not sufficient to update information. However, the traversal tree that is generated during the reachability computation, or a variant of it, may be used to update reachability information. Thus, we overcome the aforementioned problem by storing a variant of the traversal tree that can be generated during reachability computations. We call this variant the reached state relation ($P(x, y)$). This variant must satisfy the following properties:

- It must be acyclic.
- It must only include edges between two reachable states.
- For each reachable state, $P(x, y)$ must contain a path from one of the initial state to it.

Any relation that satisfies these conditions is a valid representation of the reached states.

Note that if $P(x, y)$ is any acyclic relation (graph) then $\exists_y P(x, y) + \exists_y P(y, x) = R(x)$. If $P(x, y)$ is a tree we say that P is a spanning tree.

We implement two incremental algorithms. The first chooses P to be the spanning tree that is obtained by retaining only one of the many edges traversed to a reach a state from one of its neighbors. The second chooses P to be a spanning graph that is a subset of the transition relation. Figure 5 shows the spanning tree, and the spanning graph for a given transition structure. Thus, instead of storing the reachable states $1, 2, 3, 4, 5$, we store the tree $(1, 2), (2, 3), (3, 4), (1, 5)$, or the graph $(1, 2), (2, 3), (3, 4), (1, 5), (5, 3)$.

Note that for these two representations, $I(x) + \exists_y P(y, x) = R(x)$, also holds. Once the designer changes the system, the current $P(x, y)$ is modified using information about the changes made to the system and this process is repeated as often as the system changes.

4.1 Characterizing Incremental Changes

There are four different incremental changes to an instance of reachability. Briefly, changes to the system may consist of 1) addition or subtraction of edges to the transition relation, and 2) addition or subtraction of states (and hence edges) to the state space of the machine. Addition and subtraction of states can be characterized in terms of edges. Removing a state from the reached state set is equivalent (behaviorally) to removing all edges to the state, thus making it unreachable. Similarly, if a state is added to the reached state set, it is similar to making one of the unreachable states (with no edges) in the state space reachable by adding edges.

Thus, we consider only two types of incremental change: addition and subtraction of edges. For each type we first deal with a set of changes of the same type, and then we provide a general incremental algorithm to handle a complex change with many individual types. The algorithms are given in terms of implicit BDD operations.

Suppose the designer modifies the original transition relation T to a new transition relation T^{new}. Using T^{new} and T, we create two sets: sub and add. sub consists of all deleted transitions, which were removed in T^{new} and add consists of all transitions added in T^{new}. The exact computation of add and sub under different methods for changing input, is described in the following section.

4.2 Computing add and sub using T^{new}

Modifications can be made to the system at many levels; the designer may input the changes in a high-level language like Verilog. Alternately the internal data-structures of the algorithm can be directly modified. In our system, the designer is allowed to directly modify the component transition relations, or to input new processes to the system. The designer is allowed to modify the finite state machine in one of two ways: Either sets of edges are added and subtracted directly from the transition relation (or component machine transition relations), or new component machines are added to the original to modify it. In the first case the computation is self evident; add is merely the set of added transitions, i.e. $add = T^{new} \cap \overline{T}$, and sub is the set of subtracted transitions, i.e. $sub = T \cap \overline{T^{new}}$. In the second case, the state space of the original transition relation must be expanded to account for the new processes, and hence if $T^{new} = T \times T_1$, where T_1 is the transition relation of the added component, then $add = T^{new} \cap \overline{(T \times S_1)}$, and $sub = (T \times S_1) \cap \overline{T^{new}}$, where $S_1 = R_1(x) \times R_1(y)$, and $R_1(x)$ are the reachable states of T_1.

4.3 Spanning Tree Incremental Algorithm

In this section we deal with an incremental algorithm, which chooses $P(x, y)$ to be a spanning tree that can generated during the course of reachability computations.

4.3.1 Computing the spanning tree

The implicit reachability algorithm described in Section 3, begins with a current set of the initial states of the FSM. At each stage the set of states reachable in one step from the current set are computed, and added to the current set. This set of states that can be reached in one step is called the “image”. Computing the image of the current set involves computing the edges of the FSM that begin at any state in the current set, and terminate at any state of the FSM. This is part of a BFS traversal of the state

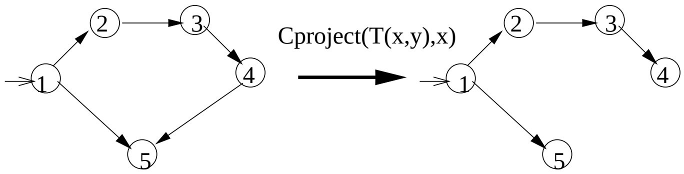

Figure 4: Using Cproject

transition graph. During this BFS procedure we choose to select only one of the many edges that terminate at a given state. This returns a spanning tree graph that spans all the reachable states of this FSM. In order to decide which edge to choose as the representative edge, any selector function like "cproject" may be used.

- **Cproject Operator**[12]: The cproject operator, when supplied a general acyclic graph, can be used to extract a tree graph that is a subset of the original input graph. The cproject operator is a selection operator, which when given a relation $T(x,y)$, and a reference vertex $\alpha(y) = \alpha_1, \ldots, \alpha_n$ in the $y = y_1, \ldots, y_n$ variables, $F = cproject(T(x,y), \alpha(y)) = \{(x,y) | T(x,y) = 1 \text{ and } \forall y', T(x,y') \Rightarrow y$ is closer to α than $y\}$. For a fixed $\alpha = (0 \ldots 0)$, the cproject operator can be computed using the recursive formulation $cproject(T(x,y), y_1 \ldots y_n) = y_1 \cdot cproject(T_{y_1}, y_2 \ldots y_n) + \overline{y_1} \cdot (\exists_y T_{y_1}) \cdot cproject(T_{\overline{y_1}}, y_2 \ldots y_n)$. For example, the operation of cproject is shown in Figure 4.

Thus the spanning tree is computed by the following algorithm, where $P(x,y)$ denotes the spanning tree, $R(x)$ the set of reachable states, and $\tau_0(x,y)$ is the initial spanning tree.

The following algorithm takes as input a subset of the spanning tree (which can be the empty tree), and returns a spanning tree for the reachable states in the FSM.

Algorithm 4.3.1$(T(x,y), \tau_0(x,y))$

$f(Q(x,y)) = cproject(R(x) \cdot T(x,y) \cdot \overline{R(y)}, x) + Q(x,y)$

where $R(x) = \exists_y Q(y,x) + I(x)$

$P(x,y) = LFP(f(), \tau_0(x,y))$

return $P(x,y)$

As an example of this procedure consider the example in Figure 5

Lemma 4.1 *Algorithm 4.3.1 is correct, i.e. it returns a spanning tree of the state transition graph if an empty tree is supplied* $\tau_0(x,y) = \phi$.

A stronger statement about this algorithm for the spanning tree, can be stated as follows:

Theorem 4.2 *The Algorithm 4.3.1 returns a correct spanning tree, if* τ_0 *is any subset of the spanning tree.*

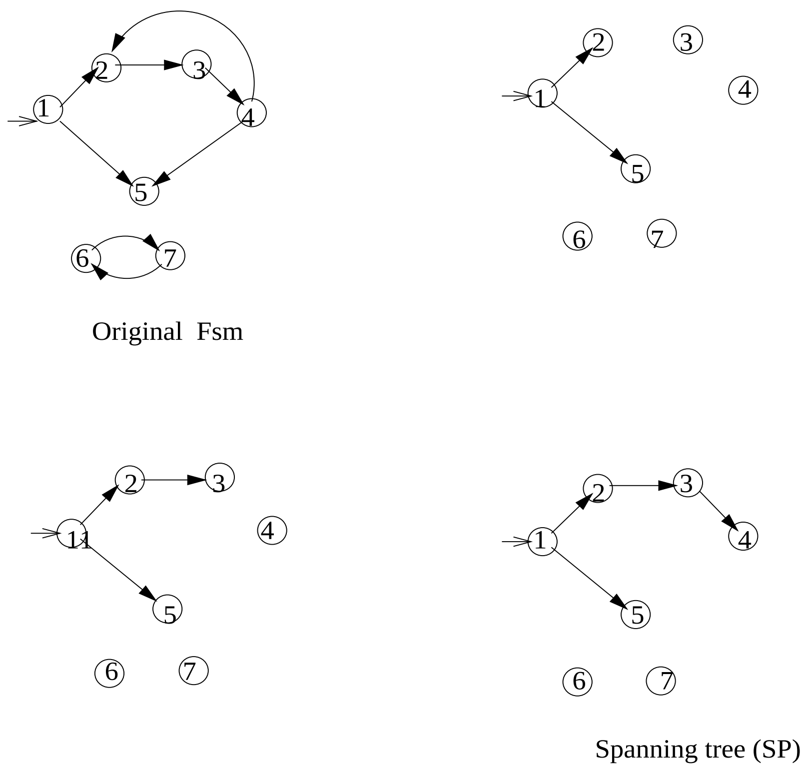

Figure 5: Computing the Spanning Tree $P(x, y)$

4.3.2 Addition of Edges

If the only changes to the system consist of the addition of edges to the transition relation, then the new spanning tree is a superset of the current spanning tree. Note that adding edges to the transition relation can never make a reachable state unreachable and hence can never remove a state (representative edges) from the spanning graph. Hence the new spanning tree must be a superset of the current spanning tree. The following lemma summarizes this:

Lemma 4.3 *If the only change to the system consists of the addition of edges to the transition relation, then* $P(x, y) \subseteq P^{new}(x, y)$.

4.3.3 Deletion of Edges

If edges that do not belong to the spanning tree are deleted from the transition relation, they do not affect the spanning tree, and it remains the same. However, if these edges do belong to the spanning tree, then potentially every (eventual) successor edge of each deleted edge may be removed from the spanning tree. After the removal of these edges, we may be left with a proper subset of the spanning tree. This is the starting point for the iterative reachability Algorithm 4.3.1.

Let $sub(x, y)$ denote the edges that are deleted from the transition relation, and $P^{+}(x, y)$ denote the spanning tree minus $sub(x, y)$ and all successors. $P^{+}(x, y)$ may be computed as the greatest fixed point of

$P^+(x,y)\cdot(\exists_y P^+(y,x)+I(x))$, given $P(x,y)$; i.e. by iteratively deleting all states that have no predecessors. This notion is formalized in the following lemma:

Lemma 4.4 *If the only change to the system consists of the subtraction of edges from the transition relation,* $f(Q(x,y)) = (Q(x,y)\cdot(\exists_y Q(y,x)+I(x)))$*, and* $P^+(x,y) = GFP(f(), P(x,y)) \subseteq P^{new}(x,y)$.

4.3.4 Incremental Spanning Tree P Algorithm

A general change consists of both the addition, and subtraction of edges. Let T^{new} denote the new transition relation that is obtained by adding, and subtracting the requisite edges from the transition relation, and P^{new} be the corresponding spanning tree. Lemmas 4.3 and 4.4 can be combined to give the following lemma.

Lemma 4.5 *For any general change to the system,* $f(Q(x,y)) = (Q(x,y)\cdot(\exists_y Q(y,x)+I(x)))$*, and* $P^+(x,y) = GFP(f, P(x,y)) \subseteq P^{new}(x,y)$.

Note that this lemma, in conjunction with Theorem 4.2 can be used to compute a new spanning tree via the following algorithm,

Algorithm 4.3.4($T(x,y), P(x,y), I(x), sub(x,y), add(x,y)$**)**

$T^{new}(x,y) = T(x,y) + add(x,y) - sub(x,y)$

$f(Q(x,y)) = (Q(x,y)\cdot(\exists_y Q(y,x)+I(x)))$

$P^+(x,y) = GFP(f, P(x,y))$

return Algorithm 4.3.1($T^{new}(x,y), P^+(x,y), I(x)$)

Here $P(x,y)$ denotes the spanning tree before the change, T^{new} is the new transition relation, and $I(x)$ the initial set of states. In order to demonstrate this algorithm consider Figure 6.

4.4 Incremental Spanning Graph Algorithm

Because the computation of the spanning tree is somewhat complicated, a variant of this procedure, which computes a spanning graph rather than a tree may also be used. The basic procedure is the same; however this procedure does not use the cproject selector, as it does not require a tree. This graph $P'(x,y) \subseteq T(x,y)$ is computed as shown, where $\tau_0(x,y) = I(x)\cdot T(x,y)\cdot\overline{I(y)}$:

Algorithm 4.4($T(x,y), \tau_0(x,y)$**)**

$f(Q(x,y)) = (R(x)\cdot T(x,y)\cdot\overline{R(y)}) + Q(x,y)$

where $R(x) = \exists_y Q(y,x) + I(x)$

$P'(x,y) = LFP(f(), \tau_0(x,y))$

return $P'(x,y)$

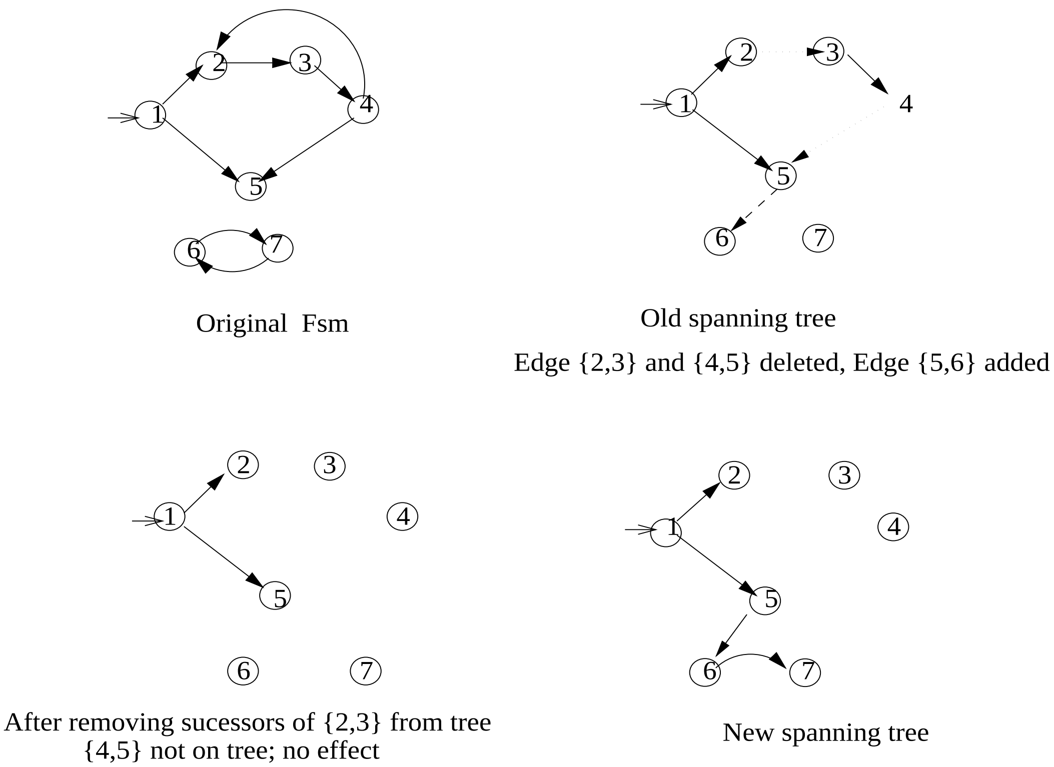

Figure 6: Updating the Spanning Tree $P(x, y)$

4.4.1 Addition of Edges

All conclusions that were made for a spanning tree in the previous section, also hold for the spanning graph. Hence Lemma 4.3 also holds for the graph $P'(x, y)$.

4.4.2 Deletion of Edges

If edges that do not belong to the spanning graph are deleted from the transition relation, they do not affect the spanning graph, and it remains the same. However, if these edges do belong to the spanning graph, then potentially every (eventual) successor edge of each deleted edge may be removed from the spanning graph. Note that any successor that has another predecessor does not need to be removed. After the removal of these edges, we may be left with a proper subset of the spanning graph. This is the starting point for the iterative reachability Algorithm 4.4. This set can also be computed by using lemma 4.6, i.e. retaining states that have predecessors.

Lemma 4.6 *If the only change to the system consists of the subtraction of edges from the transition relation,* $f(Q(x, y)) = (Q(x, y) \cdot (\exists_y Q(y, x) + I(x)))$*, and* $P^+(x, y) = GFP(f(), P(x, y)) \subseteq P^{new}(x, y)$*.*

4.4.3 Incremental Spanning Graph P' Algorithm

A general change consists of both the addition, and subtraction of edges. Let T^{new} denote the new transition relation that is obtained by adding, and subtracting the requisite edges from the transition relation, and P'^{new} be the corresponding spanning graph is computed via the following algorithm,

Algorithm 4.4.3($T(x,y), P'(x,y), I(x), sub(x,y), add(x,y)$**)**

$T^{new}(x,y) = T(x,y) + add(x,y) - sub(x,y)$

$f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$

$P'^{+}(x,y) = LFP(f, P'(x,y))$

return Algorithm 4.4($T^{new}(x,y), P'^{+}(x,y)$)

Here $P'(x,y)$ denotes the spanning graph before the change, T^{new} is the new transition relation, and $I(x)$ the initial set of states.

5 Results

We have implemented the algorithms described in the previous sections, in the HSIS [13] environment, and tested these on some ISCAS 89 benchmarks. Each example was modified, and the new reachable set was computed for a general change consisting of both addition and subtraction of a large set of edges. This is iterated many times. The actual set of edges that are added, and subtracted is randomly chosen.

The following graph (Figure 7 and Figure 8) summarizes the results. The basic algorithm was run once, and then random changes consisting of addition and subtractions of sets of edges, were made. After these changes were made, both incremental and non-incremental algorithms were run on the new input. This was iterated many times. The total times summed over 3 successive iterations is reported in the graph. All implementations used implicit formulations and the BDD data structure. The NR algorithm used was the non-incremental Algorithm 3 reported in section 3. IST refers to Algorithm 4.3.4, and ISG refers to Algorithm 4.4.3; both are reported in section 4. Unfortunately, in our experiments the changes were made at random, and so the NR algorithms reported in Figure 7 and Figure 8 do not report the results for the same set of changes. All successive incremental changes were made directly to the system within the HSIS environment.

We report the results for a few sample representative examples from the set of benchmarks. The results show that for small changes, the incremental algorithm was can be much faster than the non-incremental algorithm, and as the size of the example is increased so do the potential gains. It is important to note that the incremental algorithm does actually take more time than the non-incremental algorithm for certain changes and examples.

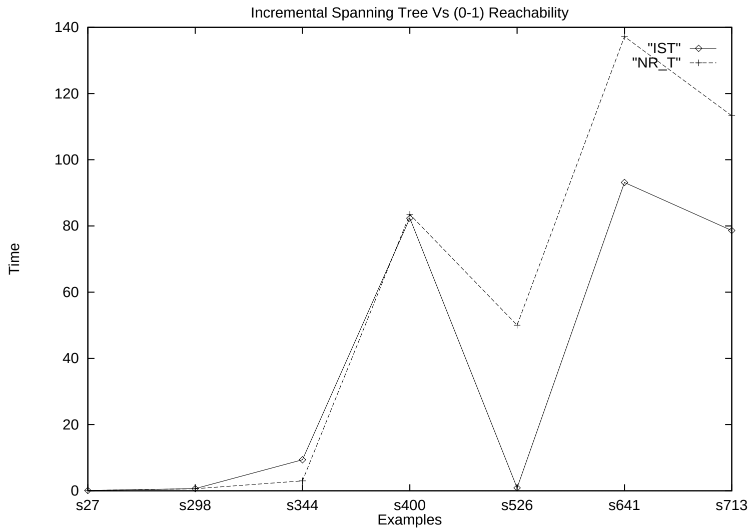

Figure 7: Results IST Vs. NR

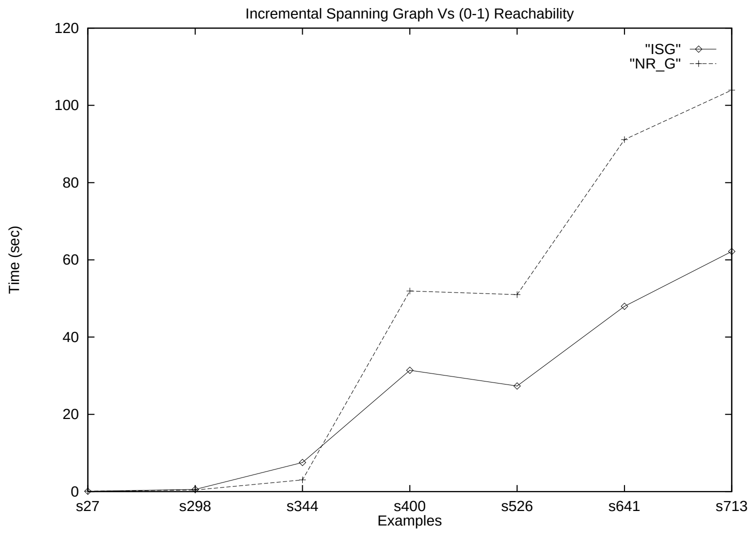

Figure 8: Results ISG Vs. NR

6 Conclusions and Future directions

We have described, and implemented two incremental algorithms for reachability. The results on a few benchmark circuits are favorable. Currently we are in the process of refining our implementation to run it on larger and more complex circuits to examine the efficacy of our methods. We also intend to try using our methods to compute the reachable set of hard examples where the normal reachability computation might blow up, by successively refining the input. In addition, we are also examining alternate representations for the reachable states.

It is important to note that since these methods store more information, memory requirements may pose a problem.

We are planning to use our incremental algorithms to create more efficient verification and synthesis methods.

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] 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.

[7] G. M. Swamy, V. Singhal, and R. K. Brayton, "Incremental methods for Fsm Traversal," Tech. Rep. UCB/ERL M95/, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1995.

[8] E. A. Emerson, "Temporal and Modal Logic," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072, Elsevier Science, 1990.

[9] G. M. Swamy and R. K. Brayton, "Incremental Formal Design Verification," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 458–465, Nov. 1994.

[10] K. L. McMillan, *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[11] G. Ramalingam and T. Reps, "On the Computational Complexity of Incremental Algorithms," Tech. Rep. TR 1033, University of Wisconsion, Madison, University of Wisconsion, Madison, 1991.

[12] B. Lin and F. Somenzi, "Minimization of Symbolic Relations," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 88–91, Nov. 1990.

[13] R. Brayton et al., "HSIS: A BDD-Based Environment for Formal Verification," in *Proc. of the Design Automation Conf.*, pp. 454–459, June 1994.