Incremental Methods for FSM Traversal

Gitanjali M. Swamy **Robert K. Brayton** **Vigyan Singhal**
Department of Electrical Engineering and Computer Sciences. University of California at Berkeley Berkeley, CA 94720
Cadence Berkeley Labs 1919 Addison St. Berkeley, CA 94704

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. The process of design is usually iterative, and the designer may modify and recompute information many times, and reachability is called each time the designer modifies the system, because current methods for reachability analysis are not incremental. Unfortunately, the representation of the reachable states that is currently used [1] in synthesis and verification, 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. This results in computational savings, as demonstrated by the results

1 Introduction

Reachability is an essential computation in both formal verification [2] as well as sequential synthesis [3] & [4]. 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 (nodes) 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 a binary decision diagram or BDD [5], is sometimes used to represent all the required quantities. e.g. the transition relation, which implicitly represents the FSM's state transition graph, and any set of states (initial, reachable etc.). When BDD's are used, the steps in the BFS traversal of the state transition graph, can be written as fixed point computations of propositional formulae on the transition relation, initial states etc ([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.

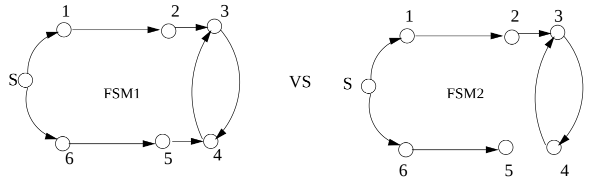

Figure 1: Reachability is non-incremental

We overcome this problem by storing a reached state relation to represent the reachable states, instead of the set of reached states. The spanning tree, or graph that can be generated by any BFS type procedure for reachability are valid reached state relations. We update this relation after every change. 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. We store the spanning tree of edges computed, during FSM traversal. We use information about the changes made to the system, and the original spanning graph of the reachable states, to compute a new spanning tree, which represents the new reachable states.

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 algorithms for reachability; how this information can be computed, and updated in an implicit framework using BDD's. We present some results in section 5. We present future directions for this work in section 6.

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 $\gamma \in \Gamma$.

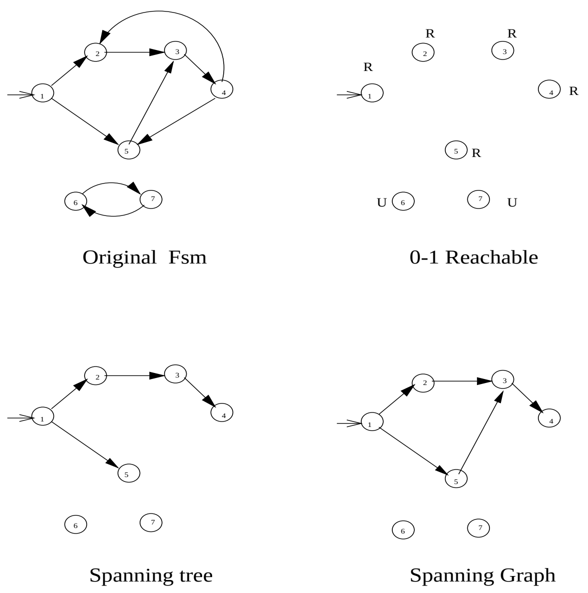

Figure 2: Different ways of storing the reachable states

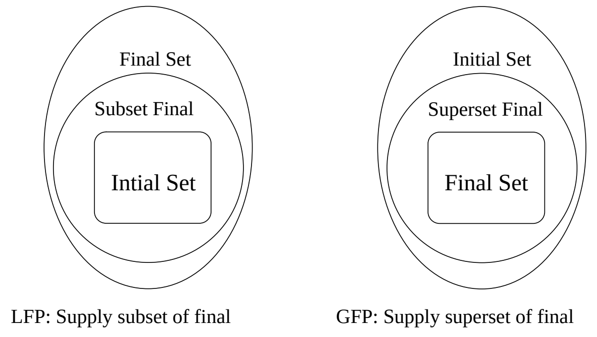

Figure 3: Fixed Points

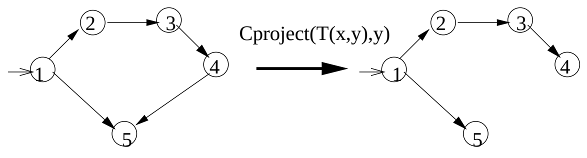

Figure 4: Using Cproject

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, $r = r_o \ldots r_i \ldots, r \in Q^\omega$, 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 Reachable states: *The set of reachable states is denoted by R, $q \in R$ if and only if there is a path from some initial state $q_0 \in I$ (the set of initial states) to q.*

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

Definition 3 Fixed point *Let $f(x)$ be a monotone (increasing or decreasing) function, the fixedpoint FP of f given I is given by the set f^i, where $f(f^i(I) = f^i(I))$ (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.

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, when supplied to any algorithm to compute it. Figure 3 illustrates this point. A similar statement can be made about GFP computations, as shown by Swamy, and Brayton [9].

Definition 4 Cproject Operator*[10]: The cproject project operator can be used to extract a tree subset graph of an acyclic graph. The cproject operator is a selection operator, which when given a relation $T(x,y)$, and a reference vertex $\alpha(y) = \alpha_1(y_1), \ldots, \alpha_n(y_n)$ in the $y = y_1, \ldots, y_n$ variables, $F = cproject(T(x,y), y) = \{(x,y') | \forall x \exists y'\ s.t.\ y' = closest\ vertex\ to\ \alpha\ such\ that\ T(x,y') = 1\}$.*

The interested reader may refer to [10] for a more detailed description of the cproject operator. For example, the operation of cproject is shown in Figure 4.

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 [6] Burch et-al [11], and Coudert et-al [1] independently extended this concept to handle reachability in larger systems, by using implicit methods. All quantities (transition relations, sets of states etc.) are represented by BDD's (binary decision diagrams), and the algorithm is represented by a fixed point computation (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. 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.

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 the reachable set. However, the traversal tree that is generated during the fsm traversal, or a variant of it, may be used to update the reachable set. 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, there must be 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, during reachability computations. 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 state space is equivalent (behaviorally) to removing all edges to the state, thus making it unreachable. Similarly, if a state is added to the state space, it is similar to making one of the unreachable states 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 [7].

4.2 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.

4.2.1 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 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" (defined in Section 2) may be used. 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 starting spanning tree (which can be the tree of edges from initial states), and returns a spanning tree for the reachable states in the FSM.

```
Algorithm 4.2.1($T(x, y)$, $\tau_0(x, y)$)
    $f(Q(x, y)) = cproject(R(x) \cdot T(x, y) \cdot \overline{R(y)}, y) + Q(x, y)$
    where $R(x) = \exists_y (Q(y, x) + Q(x, y))$
    $P(x, y) = LFP(f(), \tau_0(x, y))$
  return $P(x, y)$
```

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

Lemma 4.1 *Algorithm 4.2.1 is correct, i.e. it returns a spanning tree of the state transition graph if $\tau_0(x, y) = cproject(I(x) \cdot T(x, y) \cdot \overline{I(y)}, y)$, i.e. the initial spanning tree, and $I(x)$ is the set of initial states.*

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

Theorem 4.2 *The Algorithm 4.2.1 returns a correct spanning tree, if τ_0 is any subset of the spanning tree that includes all the initial states .*

4.2.2 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 $P(x, y) \subseteq P^{new}(x, y)$.*

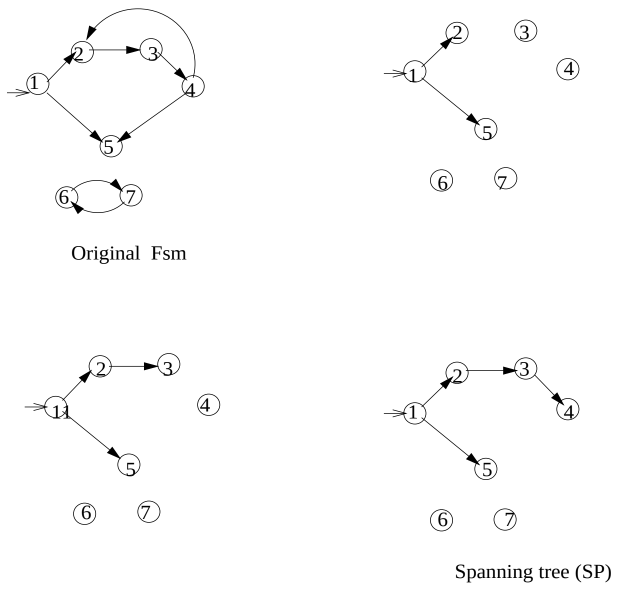

Figure 5: Computing the Spanning Tree $P(x, y)$

4.2.3 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 4.2.1.

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 its 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 $P(x, y) - sub(x, y)$; 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* $P^+(x, y) = GFP(f(), (P(x, y) - sub(x, y))) \subseteq P^{new}(x, y)$*.*

4.2.4 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 4.2.4**
    $T^{new}(x, y) = T(x, y) + add(x, y) - sub(x, y)$
    $f(Q(x, y)) = (Q(x, y) \cdot (\exists_y Q(y, x) + I(x)))$
    $P^+(x, y) = GFP(f, (P(x, y) - sub(x, y))$
  return Algorithm 4.2.1($T^{new}(x, y), P^+(x, y), I(x)$)
```

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.

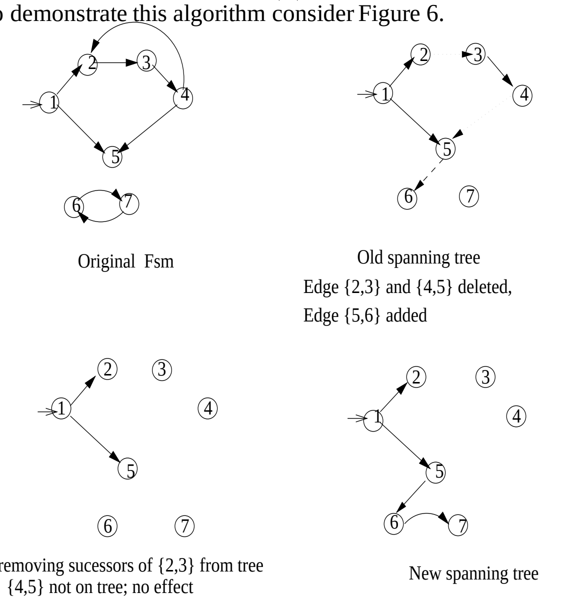
Original Fsm
Old spanning tree
Edge {2,3} and {4,5} deleted,
Edge {5,6} added
After removing sucessors of {2,3} from tree
{4,5} not on tree; no effect
New spanning tree

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

4.3 Incremental Spanning Graph Algorithm

Since 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. The computation of this graph $P'(x, y) \subseteq T(x, y)$ is given by the following algorithm, where $\tau_0(x, y) = I(x) \cdot T(x, y) \cdot \overline{I(y)}$:

```
**Algorithm 4.3**($T(x, y), \tau_0(x, y)$)
    $f(Q(x, y)) = (R(x) \cdot T(x, y) \cdot \overline{R(y)}) + Q(x, y)$
    where $R(x) = \exists_y (Q(y, x) + Q(x, y))$
    $P'(x, y) = LFP(f(), \tau_0(x, y))$
  return $P'(x, y)$
```

4.3.1 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 $P'(x, y)$.

4.3.2 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 4.3. 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* $P^{+}(x,y) = GFP(f(), (P(x,y) - sub(x,y))) \subseteq P^{new}(x,y)$.

4.3.3 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 4.3.3
    $T^{new}(x,y) = T(x,y) + add(x,y) - sub(x,y)$
    $f(Q(x,y)) = (Q(x,y) \cdot (\exists_y Q(y,x) + I(x)))$
    $P'^{+}(x,y) = GFP(f, (P'(x,y) - sub(x,y)))$
  return Algorithm 4.3($T^{new}(x,y), P'^{+}(x,y)$)
```

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.

4.4 Extending Incremental FSM Traversal to Partial Products Heuristics

All the methods described in the previous section have the intrinsic flaw that they require building the monolithic transition relation associated with the product machine. However, this is not necessary for traversal; in fact building and manipulating the monolithic product transition relation is a more time-consuming and expensive method of fsm traversal. In practise, traversal may be done using the partial product heuristics, as described in [6], [12] [13], and [14]. Thus, traversal requires computing the fixed point of

$$\begin{aligned} f(Q) &= Q(x) + \delta Q(x) \\ \delta Q(x) &= (\exists_{x,i} T_1(x,y_1,i) \ldots T_n(x,y_n,i) \cdot Q(x))_{y \leftarrow x} \\ R(x) &= LFP(f(Q), I(x)) \end{aligned}$$

where $\exists_i(T_1(x,y_1,i) \cdot T_2(x,y_2,i) \ldots T_n(x,y_n,i) = T(x,y))$, the product transition relation. However, it has been shown that it is much more efficient to find efficient methods for computing the result $f(Q)$ from the previous expression, rather than forming the product transition relation. This can be extended in order to compute the reached state relation. In this section, we will describe how the reached state relation (we will only be describing the spanning graph representation) can be computed using partial products heuristics. We will rely on the the methods of [13] to efficiently compute an expression of the form $(\exists_{x,i}(T_1(x,y_1,i) \cdot T_2(x,y_2,i) \ldots T_n(x,y_n,i)) \cdot Q(x))$ Thus algorithms 4.3 and 4.3.3 can be re-written in the partial product context as:

```
Algorithm 4.4.1($T(x,y), \tau_0(x,y)$)
    $f(Q(x,y)) = Q(x,y) + \delta Q(x,y)$
    $\delta Q(x,y) = (\exists_i T_1(x,y_1,i) \ldots T_n(x,y_n,i) \cdot R(x)) \cdot \overline{R(y)}$
    $R(x) = \exists_x Q(x,y)_{y \leftarrow x}$
    $P(x,y) = LFP(f(), \tau_0(x,y))$
  return $P(x,y)$
```

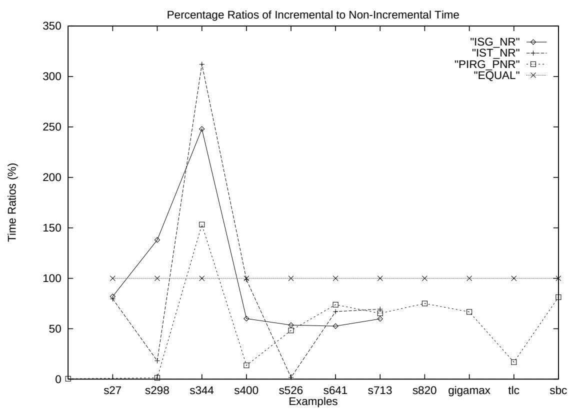

Figure 7: $\frac{IncrementalTime}{Non-IncrementalTime} \times 100$

```
Algorithm 4.4.2
    $T^{new}(x,y) = T(x,y) + add(x,y) - sub(x,y)$
    $f(Q(x,y)) = (Q(x,y) \cdot (\exists_x Q(x,y)_{y \leftarrow x} + I(x)))$
    $P'^{+}(x,y) = LFP(f, (P'(x,y) - sub(x,y)))$
  return Algorithm 4.4.1($T^{new}(x,y), P'^{+}(x,y)$)
```

For a deterministic transition system, the spanning graph, which is a subset of the transition relation is also deterministic, and hence $Q(x,i,y) = Q_1(x,i,y_1) \cdot Q_2(x,i,y_2) \ldots Q_n(x,i,y_n)$. The expressions $R(x) = \exists_{x,i} Q_1(x,i,y_1) \cdot Q_2(x,i,y_2) \ldots Q_n(x,i,y_n)$, and $\delta\ Q(x,y) = ((\exists_i (T_1(x,y_1,i) \cdot T_2(x,y_2,i) \ldots T_n(x,y_n,i) \cdot R(x)))$ are both of the form required by the heuristic algorithms of [13]. A similar extension can be made to compute the quantification operation used for the deletion of edges.

5 Results and Conclusions

We have implemented the algorithms described in the previous sections, in the HSIS [15] environment, and tested these on some ISCAS 89 and miscellaneous benchmarks. The following graphs and table(Figure 7, Figure 8 and Table 1) summarize 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 process was repeated. The actual set of edges that are added, and subtracted is randomly chosen. The NR algorithm refered to Algorithm 3 reported in section 3, IRT refers to Algorithm 4.2.4, IRG refers to Algorithm 4.3.3, and $PIRG$ refers to algorithm 4.4.2. All successive incremental changes were made directly to the system within the HSIS environment. Figure 7 plots the percentage ratio for incremental time to non incremental time for some representative examples; this is plotted for each separate method. The three ratios presented are IRG to NR, IRT to NR, and PIRG to PNR. Notice that most of the points lie below the EQUAL (100 percent) line, indicating that the incremental algorithm took less time. Only the partial product methods were able to handle larger examples, and examples tlc, gigamax, sbc etc. only report partial product times. Figure 8 presents the average ratio of the depth (number of iterations to fixedpoint within the algorithm) taken by the incremental algorithm as compared to the non-incremental algorithm in a single run of both algorithms. Notice that since this ratio is always smaller than 1, the incremental algorithm always takes fewer iterations to reach a fixed point. Table 1 reports the exact times over 3 change iterations taken by the incremental (partial product implementation) graph algorithm as compared to the non-incremental. The results

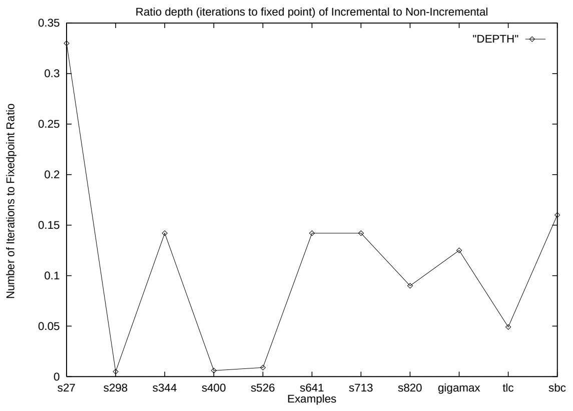

Figure 8: $\frac{IncrementalDepth}{Non-IncrementalDepth}$

Total Time (sec)		
Example	Incremental PIRG	Non-Incremental PNR
s27	0.01	0.01
s298	0.03	2.26
s344	2.76	1.80
s400	1.72	12.57
s526	3.6	7.44
s641	6.07	9.06
s713	5.78	8.85
s820	0.12	0.16
gigamax	4.07	6.11
tlc	0.09	0.53
sbc	1109.78	1363.86

Table 1: Time taken for 3 design iterations

show that the incremental algorithm can be much faster than the non-incremental algorithm. The gains come from the fewer number of iterations to fixedpoint within each algorithm, and figure 8 shows that this is indeed the case. It is important to note that the incremental algorithm may actually take more time than the non-incremental algorithm for certain changes.

6 Future directions

We have described and implemented incremental algorithms for reachability. Currently we are in the process of refining our implementation to run it on larger and more complex circuits, by storing the spanning graph as components rather than a single entity. This would also attempt to solve the memory requirement problem created because these methods store more information.

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.

We also plan to examine real design changes to come up with a more realistic way of inducing changes to the system.

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[12] J. Burch, E. Clarke, and D. E. Long, “Representing Circuits More Efficiently in Symbolic Model Checking,” in *Proc. of the Design Automation Conf.*, June 1991.

[13] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, “Efficient formal design verification data structure and algorithms,” in *Proc. Intl. Workshop on Logic Synthesis*, (Tahoe City, CA), May 1995.

[14] R. Hojati, S. Krishnan, and R. K. Brayton, “Heuristic Algorithms for Early Quantification and Partial Product Minimization,” Tech. Rep. UCB/ERL M94/11, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

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