INCREMENTAL FORMAL DESIGN VERIFICATION

by

Gitanjali M. Swamy and Robert K. Brayton

Memorandum No. UCB/ERL M94/76

30 August 1994

INCREMENTAL FORMAL DESIGN VERIFICATION

by

Gitanjali M. Swamy and Robert K. Brayton

Memorandum No. UCB/ERL M94/76

30 August 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

INCREMENTAL FORMAL DESIGN VERIFICATION

by

Gitanjali M. Swamy and Robert K. Brayton

Memorandum No. UCB/ERL M94/76

30 August 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Incremental Formal Design Verification *

Gitanjali M. Swamy **Robert K. Brayton**

Department of Electrical Engineering and Computer Science.

University of California at Berkeley

Berkeley, CA 94720

Abstract

Language containment is a method for design verification that involves checking if the behavior of the system to be verified is a subset of the behavior of its specifications (properties or requirements). If this check fails, language containment returns a subset of 'fair' states involved in behavior that the system exhibits but the specification does not. Current techniques for language containment do not take advantage of the fact that the process of design is incremental; namely that the designer repeatedly modifies and re-verifies his/her design. This results in unnecessary computation. We present a method that successively modifies the latest result of verification each time the design is modified. Our incremental algorithm translates changes made by the designer into an addition or subtraction of edges, states or constraints (on acceptable behavior) from the transition behavior or specification of the problem. Next, these changes are used to update the set of 'fair' states previously computed. This incremental algorithm takes much less time than the current techniques for language containment; a conclusion supported by experimental results presented in this paper.

1 Introduction

Design verification is the process of checking if what the designer specified is what he/she wants. One way to perform design verification on sequential logic circuits is to specify the design (also called the system), as well as, the requirements of the

*This work is supported in part by the National Science Foundation, the California Micro program, Bell Northern Research and Fujitsu design (also called the properties) as a *finite automaton* (or finite state machine), usually by the process of abstraction, and verify that the *language* (the set of behaviors) of the property is a superset of the language (or behavior) of the system. The requirement that the language of the property contains the language of the system is called *language containment*. Language containment fails due to the presence of states that show behavior that is in the system but not in the property. This set of states is called the set of 'Fair' states.

In general, the system itself need not be a single finite state machine. It is more commonly expressed as a set of interacting finite state machines that form a compound entity called the *product machine*. Figure 1 illustrates a system composed of three interacting finite state machines (M1, M2, M3), with transition relations (T_1, T_2, T_3). The transition relation of this system describes how the current state of the system and inputs relate to the next state and outputs; it is the Cartesian product of the individual transition relations of the component machines, namely $T = T_1 \times T_2 \times T_3$. The problem of language containment has to be solved in this environment of interacting finite state machines.

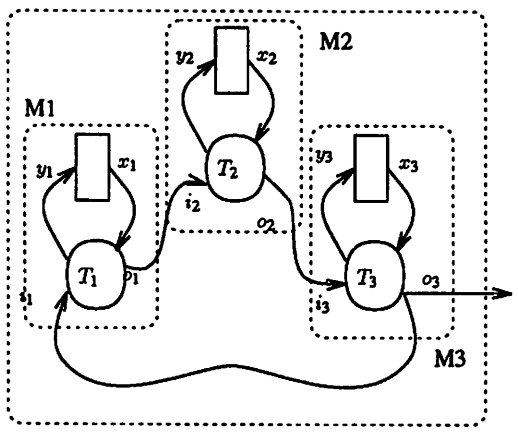

Figure 1: A System of Interacting Finite State Machines

Current techniques [1, 2] perform language containment as a single pass. If the designer modifies the design after a solution has been obtained, then the entire language containment algorithm is repeated on the new design. In practice, the process of design is *incremental*; the designer modifies and re-verifies the design many times. If standard language containment algorithms are used in real-life design situations, they often result in redundant re-computation of information because the similarity between the old system and the new system is not utilized. We introduce the concept of *incremental verification*, which allows multiple changes to the system but runs the entire language containment algorithm only once, and propagates successive changes or increments from the latest solution.

The language containment algorithms of Touati et. al.[1] and Hojati et. al. [2] start with all reachable states, and successively reduce this until only the "fair" states remain. These algorithms are monotonic in nature, i.e., once a state is removed from the set of potential fair states, it is never added back. Hence, a similar algorithm that starts with any superset of the fair set, would return the fair set. Our algorithm uses information about the change in the system and the original set of fair states to derive a smaller superset of new fair states (smaller than the set of all reachable states). Then, it reduces this superset with an algorithm similar to [2]. Since this superset is much smaller than the set of all reachable states, the incremental algorithm converges faster.

The aim of this exercise is to get the new answer to the verification decision problem, "Is what I specified what I wanted?", using the old fair states (also referred to as $Fair^{+}$), and the incremental changes that the designer made to the input problem, while spending less time and effort in this computation than if the entire language containment algorithm was run on the new problem.

The paper is organized as follows. The basic terms used in the paper are given in Section 2. Previous work is described in Section 3. Our work begins in Section 4, by recognizing that all (small) changes to the system can be translated to the addition and subtraction of edges, states and constraints. Next, we analyze how each of these changes to the system can be propagated to get the new set of fair states, also called $Fair^{+new}$ and prove the correctness of these techniques. This section summarizes how incremental changes are classified and how each particular change can be handled individually. Next, the procedures for handling individual classes of change are merged to get a general algorithm for handling any change to the system. Section 5 describes the entire incremental language containment algorithm. Finally, we conclude by presenting experimental results, which demonstrate the efficacy of this method, in Section 6, and give future direction for this research in section 7. A shorter description of this work has been published in [3].

It is important to keep in mind that all operations are to be carried out in the context of the Binary Decision Diagram (BDD) data structure[4]. Even though not explicitly stated, all sets and relations are represented as their BDD's [5].

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 directed graph, whose vertices are states, and edges are labelled with elements of $(\Sigma \times \Gamma)$. This directed graph is called a *State Transition Graph.*

Definition 2 Run: *A sequence of states, $r = r_o \ldots r_i \ldots, r \in Q^\omega$, is a run,or a* path *of T for a word $\sigma = (\sigma_0 \ldots \sigma_i \ldots)$, $\sigma \in \Sigma^\omega$, if $r_0 \in I$ and for $i \geq 0$, $T(r_i, \sigma_i, \gamma_i, r_{i+1}) = 1$. The set I refers to the set of initial states.*

The infinity set of a run r, denoted inf(r), is the set of states that are visited infinitely many times in r. A run r over T is accepting if inf(r) satisfies some acceptance condition C. The acceptance condition C distinguishes different ω-automata (Automata accepting infinite behavior; e.g. L-automata, Buchi, Streett and Rabin automata), and is used to indicate what is acceptable behavior.

The **behavior** (set of **fair runs**) of the system is a subset of the runs of the system. This subset is specified using fairness constraints on the processes of the system. The **fairness conditions** express restrictions on the infinitary behavior of the finite state machine, and are used to model the system, the environment, and acceptable behaviors. Fairness conditions are modelled differently for different classes of automata. The **language** of an automaton M, represented as $L(M)$, is the set of all strings accepted by it.

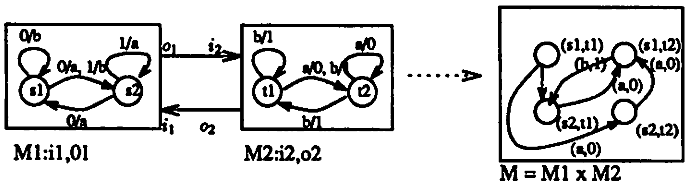

Figure 2: Forming the Product Machine

Definition 3 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.*

A *closed system* of interacting FSM's is a system with no external inputs. Any open system can be closed by adding machines that simulate the environment. Figure 2. shows a system of interacting machines M_1 and M_2, where the corresponding product machine M is indicated.

Definition 4 Language Containment: *The requirement that the language of the property (or specification) is a superset of the language of the system is called language containment.*

In the language containment paradigm, verification of the system is equivalent to determining if there is a fair path starting at an initial state. This path corresponds to behavior that is generated by the system but rejected by the task or property automaton and it is a witness to the failure of the property. The set of states which are involved in fair behavior are called *Fair states.*

Definition 5 Streett Automata *[6]: An FSM that accepts infinite behavior, which satisfies the Streett acceptance conditions is called a Streett automaton. Streett acceptance conditions consist of a finite set of ordered pairs* $C = \{(U_1, V_1), (U_2, V_2), \ldots, (U_n, V_n)$ *where* U_i *and* V_i *are subsets of the state space of the machine and run* r *is accepting if and only if* $\forall_i((inf(r) \cap U_i \neq \emptyset) \cup (inf(r) \subseteq V_i))$, *where* $0 \leq i \leq n$. *This can also be written as* $F^{\infty}(U_i) + G^{\infty}(V_i)$. *In addition, fairness constraints may also be given in the form of Positive Fair Edge* E_i *which must be traversed infinitely often and Negative Fair Edge* N_i *which must not be traversed infinitely often in any accepting run* r.

Rabin Automata [7]: The fairness conditions for a Rabin Automaton are the complements of the fairness conditions for a Streett automaton, i.e., $\forall_i((\text{inf}(r) \cap U_i = \emptyset) \cap (\text{inf}(r) \not\subseteq V_i))$. In addition, positive and negative fair edges constraints also exist.

Definition 6 L-Automata: *The L automaton[8] acceptance condition consists of a pair* $\langle R, Z \rangle$. $R \subseteq Q \times Q$, *is termed the* recur *edges, and* $Z \subseteq 2^Q$ *is the set of cycle sets. Run* r *is accepting if and only if* $\exists x \in Z, inf(r) \subseteq z$ *or* $inf_e(r) \cap R \neq \emptyset$, *where* $inf_e(r)$ *denotes the set of infinitely occurring edges in* r.

Definition 7 L-Process : *An L-process is syntactically the same as an L-automata, with one exception; the acceptance conditions for L-automata are complementary to those of L-processes, i.e. run* r *is accepting if and only if* $\forall z \in Z inf(r) \not\subseteq z$ *and* $inf_e(r) \cap R = \emptyset$, *where* $inf_e(r)$ *denotes the set of infinitely occurring edges in* r.

Definition 8 Reachable states: *The set of reachable states is denoted by R, $q \in R$ if and only if there is a path (not necessarily fair) from some initial state $q_0 \in I$ to q.*

Definition 9 Fair$^+$: *The set of states which can reach a* 'fair cycle' *(including those on it), i.e. a cycle which satisfies the fairness constraints, constitute $Fair^+$. The presence of a non-empty set $Fair^+$ indicates that the automaton has non-empty behavior.*

2.1 Some Important Computations

This section describes the computation procedure for all important operators required in this paper. 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 self same transition relation, with x and y variables interchanged (i.e. y used for present state). In general, $r(x, y)$, where r is a relation, is the same as $r(y, x)$ with x and y variables interchanged.

1. **Least Fixed Point Computation**[9]: Given an initial set of states $S_0(x)$, a transition relation $T(x, y)$, and a variable set x, the least fixed point returns a function $LFP(x, T, S_0)$ where LFP is computed as follows.

```
LFP($x, T(x, y), S_n(x)$)
$S_{n+1}(x) = (\exists x(T(x, y) \cdot S_n(x)) \cup S_n(x))_{y:=x}$
if ($S_{n+1} = S_n$)
  return $S_n$
else
  return $LFP(x, T, S_{n+1})$
```

2. **Greatest Fixed Point Computation**[9]: Given an initial set of states $S_0(x)$, a transition relation $T(x, y)$, and a variable set x, the greatest fixed point returns a function $GFP(x, T, S_0)$ where GFP is computed as follows.

```
GFP($x, T(x, y), S_n(x)$)
$S_{n+1}(x) = (\exists x(T(x, y) \cdot S_n(x)))_{y:=x}$
if ($S_{n+1} = S_n$)
```

```
    return $S_n$
  else
    return $GFP(x, T, S_{n+1})$
```

3. **Forward Reachable Operator**: Given $T(x, y)$, the transition relation and $A(x)$, a set of vertices, the forward reachable operator returns the set of vertices which can be reached by A. The forward reachable operator FR is computed using the following algorithm:

```
FR$(T, A)$
  return $LFP(x, T(x, y), A(x))$
```

4. **Backward Reachable Operator** : Given $T(x, y)$, the transition relation and $A(x)$, a set of vertices, the backward reachable operator returns the set of vertices that can reach A. It can be computed as follows:

```
BR$(T, A)$
  return $LFP(x, T(y, x), A(x))$
```

5. **Reach Reachable States Operator**: Given $T(x, y)$, the Transition Relation and $S(x)$, a set of vertices, the Reach Reachable States operator returns the set of vertices which can reach S or be reached by S. The *Reach Reachable States operator* or $RRS(T, S)$ is computed as follows:

```
RRS$(T, S)$
  return $(BR(T, S) \cup FR(T, S))$
```

6. **Forward Stable Set Operator**[2]: Given a transition relation $T(x, y)$ and a set of vertices $A(x)$, the *forward stable set operator* or $FSS(T, A)$ returns a set of states in A, which are on a cycle or can reach a cycle in A. Alternately, the FSS operator removes from A all those states which have no successors states (next states) in the transition structure. The following algorithm is used to compute the Forward Stable Set operator FSS :

```
FSS$(T, A)$
  return $(GFP(y, T(x, y), A(x)))$
```

7. **Forward Fair Path Operator**[2]: Given $T(x, y)$, the transition relation, $A(x)$, a set of states and C, a set of fairness constraints, the *forward fair path operator* or $FFP(T, C, A)$, returns a subset of states in $A(x)$ which are on a fair path. For our analysis, $C(x, y)$ are Streett fairness constraints in the form $C_i = F^{\infty}(U_i) + G^{\infty}(V_i)$ and positive fair edges $E_i(x, y)$. Hence, FFP returns those states a in A such that 1) for each E_j, there is a path in A from a to E_j, and 2) for each C_i, either $a \in V_i$ or there is a path in A from a to some state in U_i. Note that this operator returns just a path and not necessarily an infinite path. The FFP operator can be computed by using the following algorithm:

```
**FFP**$(T, C, A)$
    **return**$((\prod_{i,c_i \in C}(GFP(x, T, A \cdot U_i)) + V_i) \cap (\prod_{j,E_j \in C}(GFP(x, T, (\exists_y E_j(y, x)))))$
```

3 Previous Work

Vardi and Wolper [10] observe that the problem of verifying whether a machine (M) satisfies a given property (P) reduces to the problem of checking whether the language of the machine automaton is contained in the language of the property automaton. The language containment check in turn reduces to a *language emptiness* check for the product of the system automaton and the complement of the property automaton. Checking whether $L(M) \subseteq L(P)$ is the same as checking whether the language of $D = M \times \overline{P}$ is empty, i.e., whether $L(M \times \overline{P}) = \phi$.

When P is expressed as an L-automaton, the problem of complementing P is solved by expressing it as an L-process [1]. The acceptance conditions for L-processes and L-automata are complementary and representing $\overline{P}$ by a L-process is easily done (if P is deterministic) by just keeping the same transition structure and complementing the acceptance conditions (the complementation is implicit by the choice of representation). Similarly when P is expressed as a Rabin automaton the problem of complementation is solved by expressing $\overline{P}$ as a Streett automaton, since the acceptance conditions for Rabin and Streett automata complementary. Our experiments use a Streett and Rabin environment, and hence all successive discussions in this report are centered around Streett and Rabin automata, but are also applicable to other classes of automata.

A *language emptiness* check remains to be done, and it is performed by checking the product automata $D = M \times \overline{P}$ for acceptable infinite behavior[1] (or fair paths), which indicate that the language for the system-property product machine is not empty. A cycle is associated with any infinitary behavior in a finite graph, and in order for this infinite behavior to be acceptable, this cycle must also satisfy the fairness constraints. Thus, a machine has a non-empty language if there exists a path from an initial state to an accepted cycle, i.e., the cycle satisfies the fairness constraints specified in the automaton. The set of states that lie on such cycles form a set of *fair states* or states, which cause the *fair or non-empty* behavior. This set is also called the set of *Fair* states. In general, we compute a superset of this set called $Fair^+$, which consists of all states which are on a path to a *fair cycle*.

Hojati et. al. [2] have presented an algorithm for computation of $Fair^+$, within a Streett environment. The algorithm computes $Fair^+$ by starting with the set of reachable states, and alternately applying the FSS and FFP operators. These operators successively restrict the original set of reachable states to those on a path from an initial state to a cycle (FSS) and those which are on a fair (or acceptable) path (FFP). Thus, the set $Fair^+$ is obtained by successively shrinking the set of reachable states until only those states that are on a fair path from some initial state to a fair cycle remain. The algorithm for verification in the Streett-Rabin environment becomes:

```
Algorithm 3.1: Non_Incremental_Language_Containment
  $Fair^+$ = Compute_Fair$^+$
  if $Fair^+$ is empty return(PASS)
  else return(FAIL)
```

The set $Fair^+$ is computed using the following algorithm:

```
Algorithm 3.2: Compute_Fair$^+$
  Restrict the Transition Relation $T(x, y)$ to reachable states
  Remove negative fair edges
  Set $S_0$ = Reachable states
  While $S_{n+1} \neq S_n$
      $S' = FSS(T, S_n)$
      $S_{n+1} = FFP(T, C, S')$
  return $S_n$
```

The proof of correctness of this algorithm can be found in [2].

This algorithm has a complexity of $O(N^2)$, where N is the number of reachable states in the state space. At each iteration of the fixed point computation, at least one state in the set of reachable states, but not in $Fair^+$ is deleted from the reachable set, and this step takes $O(N)$ time, which results in an overall complexity of $O(N^2)$. This computation of complexity assumes that each step takes $O(1)$ time, and all sucessive arguments on complexity in this paper, also make this assumption. Even, if this assumption did not hold, the complexities are valid for comparing the incremental method to the non-incremental method.

Though not explicitly stated in the above algorithm, the set of reachable states can also be used to minimize the transition relation BDD. This simplification results in a considerable speedup and will be used throughout this paper without an explicit mention.

We require that the incremental algorithms in this paper take less time than the corresponding non-incremental algorithm. This criterion differs from the work done by Ramalingam etal.[11], which studied incremental algorithms for certain graph problems. This paper defined incremental algorithms as those algorithms, whose time complexity could be written as a function of the change to the system alone, where the change or Δ could be written as the sum of the change in the input and output of the algorithms. Thus, $\Delta = \Delta_{input} + \Delta_{output}$, and the complexity of the algorithm $= O(f(\Delta))$. They showed that some problems were intrinsically non-incremental; i.e. there was no locally persistent (storing only local information) algorithm that could be written for updating the information, which had a complexity only dependent on the size of the change. One such problem is the problem of reachability, which is essentially the heart of all verification algorithms. Hence, for our purposes, we will impose a slightly less rigid criterion for incrementality, by only requiring that the incremental algorithm take less time than the corresponding non-incremental algorithm.

4 Incremental Language Containment

4.1 Overview

The computation of $Fair^+$ involves successive applications of the FSS and FBP operators, which involve the successive reduction of the set of states involved. It is important to note that the algorithm begins with a superset of the states in $Fair^+$ (namely all reachable states), and eliminates states. Once a state has been removed from this set, it is never added back, and hence the algorithm is monotonic. Our incremental algorithm is based on the fact that if any superset of $Fair^+$ is given to Algorithm 3.2, it still returns the set $Fair^+$. The trick lies in using the previously computed $Fair^+$ and the changes to the system to obtain a superset of the new $Fair^+$, which is not necessarily as large as the set of all reachable states, and in most cases is significantly smaller. Given a smaller set, the algorithm converges faster and hence the incremental algorithm is typically faster.

4.2 Characterizing Incremental Changes

Recall that $Fair^+$ is a set of states that characterize the fair or unwanted behavior in the system. We want to use information about the changes to the system to incrementally modify $Fair^+$. The potential for speedup in this method is that $Fair^+$ need not be recomputed from the beginning; intermediate results can be used to avoid unnecessary computations.

At the start we proceed normally, running language containment to obtain $Fair^+$. Once the designer changes the system, the current $Fair^+$ is modified using information about the changes made to the system and this process is repeated as the system changes.

We have categorized six different incremental changes to an instance of the language containment problem. Briefly, changes to the system may consist of 1) addition or subtraction of edges to the transition relation, 2) addition or subtraction of states (and hence edges) to the state space of the machine and 3) addition or subtraction of fairness constraints. Addition and subtraction of states can be characterized in terms of edges. Clearly, 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 four types of incremental change: addition and subtraction of edges and constraints. 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 algorithm is 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 T^{sub} and T^{add}. T^{sub} consists of the original transition relation T minus all transitions, which were removed in T^{new} and T^{add} is T^{sub} plus all the transitions added in T^{new}. Note that $T^{add} = T^{new}$, but for the purposes of incremental modification we can deal with T^{add} as a single modification to T^{sub} by only adding edges. The exact computation of T^{add} and T^{sub} under different methods for changing input, is described in Section 5.

Note that fairness constraints never affect the transition structure; they only affect the FFP operator (Section 2). The new fairness constraints, with constraints added and subtracted from the original set, are used to compute a new FFP operator.

4.3 Subtraction of Edges

Consider the system obtained after subtracting a set of edges from the transition relation. Subtracting an edge cannot make any unreachable state reachable, nor can it create a new cycle in the state transition graph. Thus, subtracting an edge can never add a new state to $Fair^+$. Figure 3 indicates that deleting edge ab can potentially remove all states in sets A and B from the

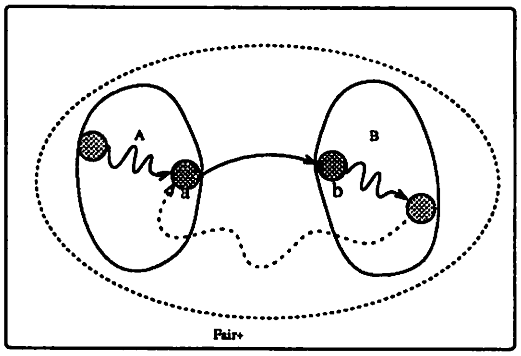

Figure 3: Deleting edge ab can potentially remove all states in A and B from $fair^+$

set $Fair^+$. The following lemma formalizes this idea.

Lemma 4.1 *The set $Fair^{+new}$ for the new system obtained by deleting edges from the original transition relation is a subset of the $Fair^+$ of the original system.*

Proof Assume the converse. $\exists_s (s \in Fair^{+new}), (s \notin Fair^+)$. Hence $s \in R^{new}$, where R^{new} is the new reachable set, and s can reach a new fair cycle. Since transitions have only been deleted from the transition structure no states can be made reachable and no new cycles can be added. Hence, s was reachable in T (the original transition structure) and s could reach a fair cycle. This contradicts $s \notin Fair^+$. ■

Thus, if the only change induced in the system consists of subtraction of edges from the state transition graph, then the following algorithm can be used to generate $Fair^{+new}$ given the new transition relation and the old set of states comprising $Fair^+$.

```
Algorithm 4.3($T^{sub}, C, Fair^+$)
  $InitSts$ = Initial States
  $R^{sub} = FR(T^{sub}, InitSts)$
```

```
**Remove Negative Fair Edges from** $T^{sub}$
    $Fair_0 = Fair^+ \cap R^{sub}$
**While** $Fair_{n+1} \neq Fair_n$
    $A = FSS(T^{sub}, Fair_n)$
    $Fair_{n+1} = FFP(T^{sub}, C, A)$
**return** $Fair_n$
```

Theorem 4.2 *If the only changes induced in the system consist of subtraction of edges from the state transition graph then Algorithm 4.3 is correct and returns* $Fair^{+new}$

Proof From Lemma 4.1 and Hojati et. al. [2] ■

Computing the conjunction of R^{sub}, and $Fair^+$ in step 4. of the Algorithm 4.3 is not necessary to the computation, but increases the efficiency, if the computation of R^{sub} is not expensive. For the evaluation of the complexity of this algorithm, this operation is ignored.

Subtraction of edges can only remove states from $Fair^+$. At each pass of the fixed-point computation in Algorithm 4.3, at least one state, which was in the old $Fair^+$, but not in the new $Fair^+$, is removed. Thus, it converges in at most $||Fair^+ - Fair^{+new}||$ steps. But $||Fair^+ - Fair^{+new}|| = \Delta_{output} \leq \Delta$, and each step takes $O(N)$ time. Hence, the algorithm completes in $O(N \cdot \Delta)$ time.

4.4 Addition of Edges

Consider the addition of a set of edges to the state transition graph. This may result in the creation of a new reachable cycle, whose states satisfy the fairness constraints. These states are not necessarily in $Fair^+$. Thus, addition of edges to the state transition graph may increase $Fair^+$. Figure 4 indicates that adding edge ab can potentially add all states in sets A and B to the set $Fair^+$. However, we will prove that if the addition of edges results in the addition of one or more states to $Fair^+$, these states must satisfy at least one of the following conditions in the new Transition system T^{add}:

- The state belongs to the set $Fair^+$.

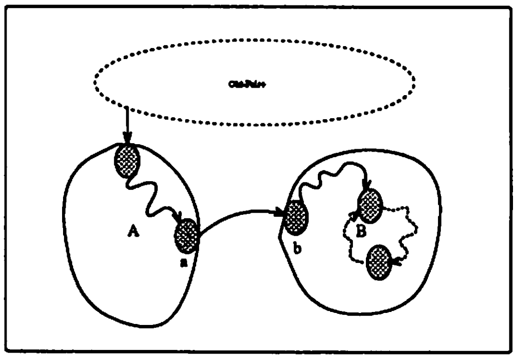

Figure 4: Adding edge ab can potentially add all states in A and B to $Fair^{+}$

- The state can reach or be reached by one of the new transitions (with negative fair edges removed). This set $Fair^{1}$ is computed as:

  $Fair^{1} = RRS(T^{add}, \exists_y T^{add}(x,y) \cdot \overline{T(x,y)})$ (Section 2).

Lemma 4.3 *The new set $Fair^{+new}$ is a subset of $Fair^{+} \cup Fair^{1}$, i.e. $Fair^{+new} \subseteq Fair^{++} \equiv Fair^{+} \cup Fair^{1}$.*

Proof Assume the converse. $\exists s (s \in Fair^{+new}), (s \notin Fair^{++})$. Hence $s \in R^{add}$ where R^{add} is the reachable set, computed using the transition structure T^{add}, and s can reach a fair cycle in the new graph T^{add}. If $s \in R$, where R is the old set of reachable states (using T) and could reach a fair cycle, then $s \in Fair^{+}$, which contradicts $s \notin Fair^{++}$. Hence, s must have been made reachable (and could reach a fair cycle) by addition of some new transition. Hence $s \in Fair^{1}$. This contradicts $s \notin Fair^{++}$. ∎

Note that $Fair^{+new} \subseteq Fair^{+}$, since the addition of edges can never remove states from $Fair^{+}$.

If the only changes to the system consist of edge addition, the new set $Fair^{+new}$ can be computed as a two step process that first computes $Fair^{++}$, and then reduces it by using the Algorithm 4.3.

```
Algorithm 4.4($T, T^{add}, C, Fair^{+}$)
  $Fair^{++} = Fair^{+} \cup RRS(T^{add}, \exists_y T^{add}(x,y) \cdot \overline{T(x,y)})$
  Remove Negative Fair Edges from $T^{add}$
  $Fair_0 = Fair^{++}$
  While $Fair_{n+1} \neq Fair_n$
    $A = FSS(T^{add}, Fair_n)$
    $Fair_{n+1} = FFP(T^{add}, A, C)$
  return $Fair_n$
```

Theorem 4.4 *If the only changes induced in the system consist of addition of edges from the state transition graph then Algorithm 4.4 is correct and returns the new set $Fair^{+new}$.*

Proof From Lemma 4.3 and Hojati et. al. [2]. ∎

As noted in Section 3, the set of reachable states can be used to simplify the BDD for the transition relation. In algorithm 4.3 the set of reachable states is not explicitly involved but may be used to simplify the transition relation BDD. It is important to note that for changes described in this section, reachability computations do not need to be carried out by starting at the initial states but need only proceed from the old set of reachable states R. This results in considerable savings in the computation.

The Algorithm 4.4 converges in at most $||Fair^{++} - Fair^{+new}|| = \Delta'$ steps. Since, $\Delta' \leq N$, for small changes, where N is the number of reachable states. Thus, the complexity of this algorithm is $O(N \cdot \Delta')$. Assuming $\Delta' \leq N$, this is faster than running the non-incremental algorithm from the beginning.

4.5 Addition of Fairness Constraints

The set $Fair^{+}$ satisfies all the fairness constraints. If new fairness constraints are only added, then the new set $Fair^{+new}$ must satisfy all of the older constraints as well as the new ones. The set $Fair^{+new}$ must be a subset of the old $Fair^{+}$.

Lemma 4.5 *If additional fairness constraints are imposed on the system, then $Fair^{+new} \subseteq Fair^{+}$.*

Proof Assume the converse, $\exists_s s \in Fair^{+new}, s \notin Fair^{+}$. Addition of constraints never affect the transition structure. Since $s \in Fair^{+new}$, s must be on a reachable path to a fair cycle. Hence, s must also be on a reachable path to a fair cycle in the old transition system. $s \notin Fair^{+} \Rightarrow \exists_c c \in C$, s violates c, where C is the old set of constraints. However, by assumption the new set of constraints $C^{new} \supseteq C$. Hence $\exists_c c \in C$, s violates c. This contradicts $s \in Fair^{+new}$. ∎

If the only change to the system consists of addition of constraints, the algorithm for computation of the new $Fair^{+new}$ is:

```
Algorithm 4.5($T, C^{new}, Fair^{+}$)
  Remove Negative Fair Edges
  $Fair_0 = Fair^{+}$
  While $Fair_{n+1} \neq Fair_n$
     $A = FSS(T, Fair_n)$
     $Fair_{n+1} = FFP(T, C^{new}, A)$
  return $Fair_n$
```

Theorem 4.6 *If the only changes to the system consist of addition of constraints then Algorithm 4.5 is correct and returns the new set $Fair^{+new}$.*

Proof From Lemma 4.5 and [2]. ∎

Using the same reasoning as Section 4.3, this algorithm has a time complexity of $O(N \cdot \Delta)$, where N is the number of reachable states.

The addition of constraints can very easily be used in conjunction with the addition and subtraction of edges. If edges are deleted, in addition to adding constraints, algorithm 4.3 can be used with the FFP operator (including the new constraints) to compute the new set $Fair^{+new}$. If edges are added then Algorithm 4.4 can be used in conjunction with the new FFP operator. The following lemmata formalize this idea.

Lemma 4.7 *If additional fairness constraints are imposed on the system, and edges are only subtracted from the transition structure then $Fair^{+new} \subseteq Fair^{+}$.*

Proof From Lemma 4.1 and Lemma 4.5 ∎

With the previous lemma, it is easily observed that the subtraction of edges and addition of constraints can be simultaneously handled by using Algorithm 4.3 (for the subtraction of edges) with the additional caveat that the FFP operator is modified to include the new constraints.

Lemma 4.8 *If additional fairness constraints are imposed on the system, and edges are only added to the transition structure then* $Fair^{+new} \subseteq Fair^{++}$*, where* $Fair^{++}$ *is as defined in Lemma 4.3.*

Proof From Lemma 4.3 and lemma4.5 ■

In a similar manner to the previous analyses, it is observed that the addition of edges and addition of constraints can be simultaneously handled by using Algorithm 4.4 (for the addition of edges) with the additional caveat that the new FFP operator (as defined in Lemma 4.5) is used for the computation.

4.6 Subtraction of Fairness Constraints

The set $Fair^+$ contains states involved in infinite behavior that satisfy all fairness constraints C_i. If some constraint $C_i = F^\infty(U_i) + G^\infty(V_i)$ is subtracted, $Fair^+$ still contains states that are involved in infinitary behavior, and satisfy all constraints $C_j \neq C_i$ (as well as C_i). Thus, the set $Fair^+ \subseteq Fair^{+new}$. In addition to the states in $Fair^+$, $Fair^{+new}$ also contains states that may be in infinitary behavior that violates the deducted constraint C_i. Such states are definitely a subset of $RRS(T, \overline{U_i \cup V_i})$; namely states that may either reach or be reached by states not in U_i or V_i. Since more than one constraint may be deleted, let $C_i, i \in S$ denoted the set of constraints to be deleted.

Lemma 4.9 *If constraints* $C_i = F^\infty(U_i) + G^\infty(V_i)$*,* $i \in S$ *are subtracted from the set of original constraints, then the set* $Fair^{+new} \subseteq Fair^{++} = RRS(T, \bigcup_{i\in S}(\overline{U_i \cup V_i})) \cup Fair^+$.

Proof If $\exists_x, x \in Fair^{+new}$, then x is involved in infinitary behavior, which satisfies all constraints $C_j = F^\infty(U_j)+G^\infty(V_j)$, $j \notin S$. The infinitary behavior that x is involved in, may or may not satisfy C_i, $i \in S$. If it satisfies all C_i, $i \in S$, then $x \in Fair^+ \Rightarrow x \in Fair^{++}$. If it does not satisfy at least one of the C_i, $i \in S$, then it must belong to infinitary behavior that violates the corresponding C_i, and it must belong to $RRS(T, \bigcup_{i\in S}(\overline{U_i \cup V_i}))$. Hence $x \in Fair^{++}$. ■

This leads to the following algorithm for changes, where constraints are only subtracted from the system.

```
Algorithm 4.6.1($T, C^{new}, S, Fair^+$)
  $Fair^{++} = RRS(T, \bigcup_{i\in S}(\overline{U_i \cup V_i})) \cup Fair^+$
  Remove Negative Fair Edges from $T$
```

```
$Fair_0 = Fair^{++}$
**While** $Fair_{n+1} \neq Fair_n$
    $A = FSS(T, Fair_n)$
    $Fair_{n+1} = FFP(T, C^{new}, A)$
**return** $Fair_n$
```

If, in addition to constraint subtraction, edges were added (added edges $= (\exists_y T^{add}(x,y) \cdot \overline{T(x,y)})$) to the transition structure, then states in the new $Fair^{+new}$ must satisfy at least one of the following conditions in the new Transition system T^{add}:

- The state belongs to the set $Fair^{+}$.
- The state can reach or be reached by one of the new transitions. This set $Fair^1$ is computed as:

  $Fair^1 = RRS(T^{add}, \exists_y T^{add}(x,y) \cdot \overline{T(x,y)})$ (Section 2).
- The state can reach or be reached by states violating the deleted constraints.

  $Fair^2 = RRS(T^{add}, \bigcup_{i \in S}(\overline{U_i \cup V_i}))$ (Section 2).

Lemma 4.10 *If fairness constraints are subtracted from the system, and edges are only added to the transition structure then* $Fair^{+new} \subseteq Fair^{++} = RRS(T^{add}, \bigcup_{i \in S}(\overline{U_i \cup V_i}) \cup (\exists_y T^{add}(x,y) \cdot \overline{T(x,y)})) \cup Fair^{+}$

Proof Assume the converse. Consider some state $s \in Fair^{+new}$. It must reach a fair cycle, and it must be reachable. If s was reachable in the old transition graph T and could reach a fair cycle, then $s \in Fair^{+}$. Hence s must be made reachable, and must reach a fair cycle by the addition of edges, or the subtraction of constraints. Hence, s must be reachable or reached by the added edges, or the subtracted constraints. But this implies that either $s \in Fair^1$ or $s \in Fair^2$, or $s \in Fair^{+}+$. ∎

If subtraction of constraints is used in conjunction with addition of edges, then the following algorithm describes the computation of the new $Fair^{+new}$.

Algorithm 4.6($T^{add}, T, C^{sub}, S, Fair^{+}$)

```
$Fair^{++} = RRS(T^{add}, \bigcup_{i \in S}(\overline{U_i} \cup \overline{V_i}) \cup (\exists_y T^{add}(x,y) \cdot \overline{T(x,y)})) \cup Fair^{+}$
Remove Negative Fair Edges from $T^{add}$
$Fair_0 = Fair^{++}$
While $Fair_{n+1} \neq Fair_n$
    $A = FSS(T^{add}, Fair_n)$
    $Fair_{n+1} = FFP(T^{add}, C^{sub}, A)$
return $Fair_n$
```

This algorithm has a complexity of $O(N \cdot \Delta')$, where N is the number of reachable states. The next section deals with putting these individual algorithms together to form a general algorithm which handles any change to the system.

5 General Algorithm

We describe an incremental algorithm for language containment, when a general set of changes consisting of deletion and addition of edges from the transition structure and addition and subtraction of constraints, is applied to the system.

We begin by separating the augmented transition relation T^{new} into T^{sub}, which consists of the original transitions relation T minus all transitions which were removed in T^{new} and T^{add} which is seen as T^{sub} plus all the transitions, which are added in T^{new}.

The change to the system can be seen as a two stage process; in the first stage, constraints are added and edges are only subtracted from the system. In the second stage constraints are only subtracted and edges are only added to the transition structure obtained from the previous stage. The first stage computes an intermediate $Fair^{+new1}$ under the assumption that the only changes consist of edge subtraction and constraint addition and for this stage, the transition structure T^{sub} is used. The second stage computes the new $Fair^{+new1}$ using as input the intermediate $Fair^{new1}$ and the new transition structure T^{add}.

5.1 Computing T^{add} and T^{sub} using T^{new} and T

Recall that, the augmented transition relation T^{new} is expressed as two separate transition relations T^{sub} and T^{add}, where $T^{add} = T^{new} = T^{sub} + (T^{new} - T)$.

The designer modifies the original transition relation T to a new transition relation T^{new} by adding and subtracting edges from the transition structure. In practice, this may be done in two ways:

1. The designer might choose to directly modify the transition structure of the original system. If $T^{add-edges}$ and $T^{sub-edges}$ represent the set of edges which are to be added and subtracted, respectively, from the system transition graph; the corresponding T^{add} and T^{sub} can be computed by using the following equations.

$$\begin{aligned} T^{sub} &= T \cap \overline{T^{sub-edges}} \\ T^{add} &= T^{sub} \cup T^{add-edges} \end{aligned}$$

2. A designer can also modify the system by adding or subtracting processes from the system of interacting processes and imposing additional constraints on these new processes. T_1 is the product transition relation of only the added processes, T^{old} is the transition relation of the original processes, and T^{new} is the new transition relation of the augmented system. R_1 is the set of states within the transition structure T_1 that are reachable from the initial states (InitSts1) in it and it can be computed as $R_1 = FR(T_1, InitSts1)$. In order to compute T^{add} and T^{sub} in this framework the following equations may be used:

$$\begin{aligned} T &= T^{old} \times R_1(x) \times R_1(y) \\ T^{sub} &= T \cap \overline{T^{new}} \\ T^{add} = T^{new} &= T_1 \cap T \end{aligned}$$

This set of equations essentially augments the old T^{old} to the product space of the Cartesian product $T^{old} \times T_1$.

Modifications can be made at many different levels. The designer may input the changes in a high level language (e.g. Verilog). Alternately, he/she might choose to augment individual subprocesses in the system of interacting processes by directly modifying the data-structure that stores their transition relations and constraints. In our implementation, the designer is allowed to directly change the individual transition relations, or input process constraints and new processes via the intermediate 'Pif' [12] format.

5.2 Incremental Language Containment

The general algorithm for computation of $Fair^{+new}$ is based on Algorithm 4.3 and Algorithm 4.4, with the additional caveat that the FFP is modified to account for the changes in the set of constraints. As described previously, the algorithm has two stages for its computation. First, the changes due to deletion of the edges and next the changes due to addition of new edges are handled. Let T^{sub}, T^{add}, and $Fair^{+}$ be defined as before, and let C^{add} be the set of constraints, which consist of the old set of constraints plus added constraints. Let C^{sub} refer to the final set of constraints; i.e., $C^{sub} = C^{add} - C_i, i \in S$. The general incremental language containment (ILC) algorithm:

```
Algorithm 5.2.1: Incremental_Language_Containment
  $Fair^{+}$ = Incremental_Compute_Fair$^{+}$
  if $Fair^{+}$ is empty return(PASS)
  else return(FAIL)
```

where the algorithm for the incremental computation of $Fair^{+}$ is:

```
Algorithm 5.2.2: Incremental_Compute_Fair$^{+}$
  $Fair^{+new1} = Algorithm\, 4.3(T^{sub}, C^{add}, Fair^{+})$
  $Fair^{+new} = Algorithm\, 4.6(T^{add}, T^{sub}, C^{sub}, S, Fair^{+new1})$
return $Fair^{+new}$
```

Theorem 5.1 *Algorithm 5.2.2 is correct and returns the new set $Fair^{+new}$.*

Proof The first stage of the algorithm does not involve addition of edges, hence the use of algorithm 4.3 is valid and returns the correct set of fair states, $Fair^{+new1}$ for this subproblem to the next stage (refer to theorem 4.2 and lemma 4.7). The second stage does not involve the subtraction of edges; hence the use of algorithm 4.6 is valid and the correct set $Fair^{+new}$ is returned (refer to Theorem 4.4, Lemma 4.8 and Lemma 4.10) ∎

6 Results

We have implemented the algorithms described in the previous section and tested these on a set of verification benchmarks. Each example was modified, and the $Fair^+$ was recomputed for a general change to the system, which consists of addition and subtraction of edges and constraints. The actual edges/constraints that were added or subtracted from the transition relation are arbitrary, and were chosen so as to make the system pass the language containment check.

The first row in each table reports the name of the example, and the iteration number. The second row reports the time taken by the incremental language containment (ILC) algorithm; this includes the time for incremental update of input data, and re-initialization. The last row reports the time for the non-incremental (NLC) algorithm with the non-incremental update; this includes the time for non-incremental input of data and initialization. The last column reports the total incremental, and non-incremental times, summed over all iterations.

We ran the incremental, and non-incremental algorithms on four examples, and made 5 successive sets of changes. The columns labelled with integers $i = 1, 2, 3, 4, 5$ report the times taken on iteration (set of changes) i. The first example, Gigamax, was a description of the gigamax distributed multiprocessor, using a shared memory architecture. The second example, Scheduler, describes a version of the scheduler example by Milner [13], and the system consists of a token ring, where element of the ring, called a cell, communicates with its "job", and its two nearest neighbor cells. The third example, Tcp, describes a simplified version of the TCP/IP communication protocol. The final example, Idlc, describes an industrial data link controller example. All the examples were written in Verilog, and translated into the *blif-mv* format using the vl2mv translator [14]. All successive incremental changes were made directly to the system within the HSIS environment.

The changes themselves were made by examining an error trace (describing some non-empty behavior in the system) generated, and deleting and adding edges and constraints so as to remove the particular error trace, and eventually make the system pass language containment.

The results show that the incremental algorithm was always considerably faster than the non-incremental algorithm.

7 Conclusions and Future

We have presented a framework for incremental language containment and shown that the incremental algorithms can be superior to non-incremental algorithms for changes in the input problem. It should be noted (from the results)that as the size

Gigamax	1	2	3	4	5	Total
ILC[1]	25.0	9.1	35.2	22.8	25.1	117.3
NLC[2]	42.4	29.1	53.9	41.1	44.9	211.5
Scheduler	1	2	3	4	5	Total
ILC	18.5	0.8	21.7	8.5	19.6	69.2
NLC	25.4	7.7	27.6	23.8	29.2	113.8
Tcp	1	2	3	4	5	Total
ILC	40.6	14.6	97.3	22.3	8.4	183.2
NLC	447.7	420.6	463.9	431.0	417.2	2180.5
Idlc	1	2	3	4	5	Total
ILC	247.1	369.8	463.2	176.6	-	1256.7
NLC	2403.2	2659.4	2461.6	2573.3	-	10094.5

Table 1: NLC Vs ILC (in seconds)

1: ILC =Incremental algorithm and incremental data update

2: NLC =Non-incremental algorithm and non-incremental data input

of the example increases (from gigamax to idlc), so does the gain from using an incremental algorithm. In addition, we are examining alternate methods for entering changes to the system. We are also searching for a set of larger benchmarks to get more evidence for the superiority of the incremental methods. Since language containment is just one approach to the problem of design verification; we intend to extend this work to model checking methods as well.

8 Acknowledgements

The authors would like to thank Felice Balarin, Adnan Aziz, and the UC Berkeley CAD group for their help. The authors would also like to thank the National Science Foundation for the support of an NSF fellowship.

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] R. Hojati, T. R. Shiple, R. K. Brayton, and R. P. Kurshan, "A Unified Environment for Language Containment and Fair CTL Model Checking," in *Proc. of the Design Automation Conf.*, (Dallas, Texas), pp. 475–481, June 1993.

[3] G. M. Swamy and R. K. Brayton, "Incremental Formal Design Verification," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 130–133, Nov. 1994.

[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] R. S. Streett, "Propositional Dynamic Logic of Looping and Converse is Elementary Decidable," *Information and Control*, vol. 54, pp. 121–141, 1982.

[7] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*, vol. 13 of *Regional Conf. Series in Mathematics*. Providence, Rhode Island: American Mathematical Society, 1972.

[8] R. P. Kurshan, *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993. To appear.

[9] 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.

[10] M. Y. Vardi and P. L. Wolper, "An Automata-Theoretic Approach to Program Verification," in *Proc. IEEE Symposium on Logic in Computer Science*, pp. 332–334, 1986.

[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] R. Hojati, V. Singhal, and R. K. Brayton, "Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment," Tech. Rep. UCB/ERL M94/12, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[13] R. Milner, *Communication and Concurrency*. New York: Prentice Hall, 1989.

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