# Incremental Formal Design Verification *

Gitanjali M. Swamy Robert K. Brayton

**Abstract**

Design verification is the process of ensuring that a designed system satisfies its original requirements. Language containment, a method for performing design verification, involves checking if the behavior of the system to be verified is a subset of the behavior of the specifications(properties or requirements). If the language containment condition is not met then the process returns a subset of states involved in behavior that the system exhibits but the specification does not. This subset is called the set of 'Bad' or 'Fair' states and they are used to indicate that the system failed verification.

In practice, the process of design is incremental and the designer repeatedly modifies and re-verifies his/her design. Current techniques for language containment do not take advantage of this, resulting in unnecessary re-computation that can be cumbersome if small changes are made to a large system. We present an alternate method, which takes advantage of the incremental nature of the design process by successively modifying the original result of verification each time the design is modified.

The first step in this *Incremental* algorithm consists of identifying the changes that the designer may make to the system or the properties to be verified. Next, every change is translated to an addition & subtraction of edges, states and constraints (on acceptable behavior) from the transition behavior or specification of the problem. These changes are used to update the set of 'Bad' states previously computed. Thus, the incremental technique successively updates the set of 'Bad' behaviors instead of recomputing the behavior after every change. This results in a faster and more efficient algorithm and makes our incremental algorithm superior to the current techniques for language containment.

*This work is supported in part by the National Science Foundation, the California Micro program, Bell Northern Research and Fujitsu

# 1 Introduction

Design verification is the process of checking whether what the designer specified was what he/she wanted. One of the ways to perform design verification on sequential logic circuits is to specify the design ( also called the system) as well as the requirements of the 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 language refers to 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*. If language containment fails, it is due to the presence of states that show behavior that is in the system but not in the property and this set of states is called the set of 'Bad' or '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 a *product machine*. Figure 1. describes a system composed of 3 interacting finite state machines. The transition relation (it describes how the current state of the system and inputs relate to the next state and outputs) of this product machine is the product of the individual transition relations of the component machines. The problem of language containment needs to be solved in this environment of interacting finite state machines.

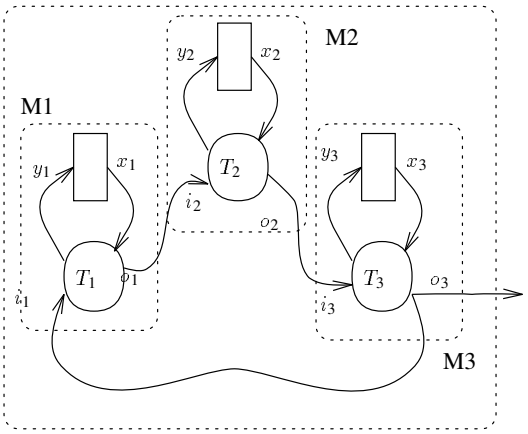


Figure 1: A System of Interacting Finite State Machines

Current techniques [7] [3] perform language containment as a single pass. If the designer modifies the design after a solution has been obtained for it, then the entire language containment algorithm is repeated for the new problem. In practice, the process of design is *incremental*; the designer modifies and reverifies the design many times. If standard language containment algorithms are used for real life design situations, they result in redundant re-computation of information because the similarity between the old system and the new system is not utilized. In order to overcome this problem, 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 first solution. Verification and language containment in particular, are decision problems. Thus,

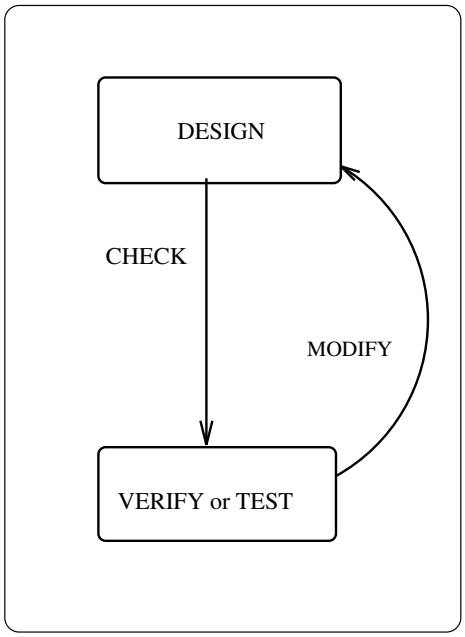


Figure 2: Design is an Incremental process

the aim of this exercise is to get the new answer to the decision problem (Is what I specified what I wanted ?) using the old solution and the incremental changes that the designer made to the input problem. In addition, we expect to spend less time and effort in this computation than if the entire language containment algorithm was run on the new problem. In general, the aim for an incremental algorithm for a design verification is getting the same yes or no answer as a non-incremental algorithm while spending less time and effort.

Our algorithm recognizes that the language containment algorithms of [8] and [3] begin by taking all reachable states and successively reduce this set until only the bad or fair states remain. An examination of these algorithms show that they are monotonic in nature; i.e. once a state is removed from the set of potential bad states, it is never added back. Hence, a similar algorithm that starts with any superset of the bad set, would return the bad (or fair) set. Our algorithm uses information about the change in the system and the original set of bad states to derive a smaller superset of bad states than the set of all reachable states. Then it reduces this superset with an algorithm similar to [3]. Since this superset is much smaller than the set of all reachable states, the incremental algorithm converges faster.

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 by recognizing that all (small) changes to the system can be translated to 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 or $Fair^{+new}$ and prove the correctness of these techniques. Section 4. 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 and section 5. describes the entire incremental language containment algorithm. Finally,

we conclude by presenting our results in section 6 and give future direction for this research in section 7.

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[1]. Even though not explicitly expressed, all quantities are represented as their BDD's [8].

## 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*
- $\Sigma$ *is a finite set of inputs*
- $\Gamma$ *is a finite set of outputs*
- $T \subset Q \times \Sigma \times \Gamma \times Q$ *is the* Transition Relation
- $I$ *is a set of initial or starting states of the machine.*

$(q, \sigma, \gamma, t) \in T$ *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[6] or a* Path *of* $T$ *for a word* $\sigma = (\sigma_1 \ldots \sigma_i \ldots)$*,* $\sigma \in \Sigma^\omega$*, 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 the acceptance condition* $C$*. The acceptance condition* $C$ *distinguishes different* $\omega$ *automata ( Automatons accepting infinite behavior).*

**Definition 3** *A* **behavior** *(a* **fair run***) 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 modeling the system, the environment, and acceptable behaviors. Fairness conditions are modelled differently for different classes of Automatons. The* **language** *of an Automaton* $M$*, represented as* $L(M)$*, is the set of all strings accepted by it.*

**Definition 4 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 inputs are those inputs to* $\{M_1, M_2, \ldots, M_n\}$ *which are not outputs of other machines, and the outputs of the product is the set of all outputs of the components. The transition relation is the intersection of the component transition relations. A* closed system *of interacting FSM's is*

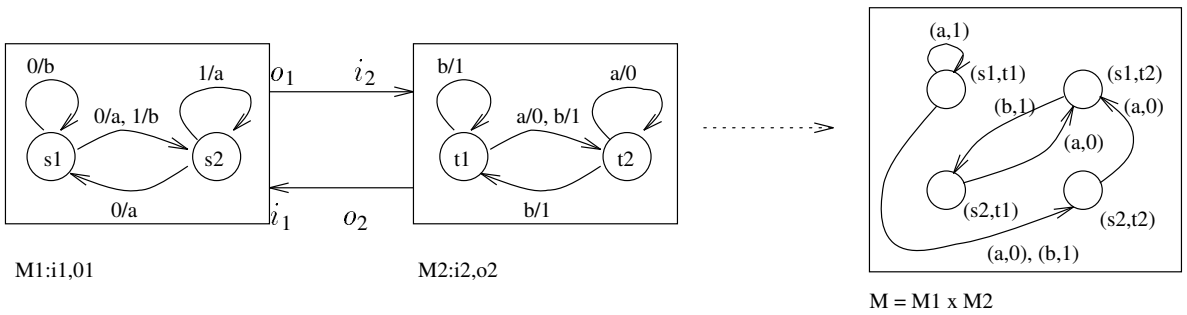


Figure 3: Product Machine

*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 $M_2$, whose corresponding product machine $M$ is indicated in figure 3.*

**Definition 5 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 automaton and it is a witness to the failure of the property. The set of states which are involved in bad behavior are called* Fair states.

**Definition 6 Streett Automaton***: An automaton which accepts infinite behavior which satisfies the Streett acceptance conditions is called a Streett automaton. The Streett acceptance condition consists of a finite set of ordered pairs $\mathcal{C} = \{(U_1, V_1), (U_2, V_2), \ldots, (U_n, V_n)\}$ where $c_i = (U_i, V_i)$ 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 $i$ ranges over a finite set. This can also be written as $F^{\infty}(U_i) + G^{\infty}(V_i)$. There are an additional class of Fairness Constraints 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 $r$.*

**Definition 7 Rabin Automaton***: A Rabin automaton is identical to a Streett automaton, in all but the fairness conditions. The fairness conditions for a Rabin Automaton are the complements of the fairness conditions for a street automaton.*

**Definition 8 L-Automaton***: The $L$ automaton[5] 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 $inf(r) \subseteq z \in Z$* or *$inf_e(r) \cap R \neq \emptyset$, where $inf_e(r)$ denotes the set of infinitely occurring edges in $r$.*

**Definition 9 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 $inf(r) \not\subseteq z \in Z$* or *$inf_e(r) \cap R = \emptyset$, where $inf_e(r)$ denotes the set of infinitely occurring edges in $r$.*

**Definition 10 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 $q_0 \in I$ to $q$.*

**Definition 11 Fair$^+$:** *The set of states which can reach or are on a* 'fair cycle' *, 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 sections describes the significance and computation procedure for all important operators required for language containment.

1. **Least Fixed Point Computation**[2]: Given an initial set of states $S_0$, a Transition relation $T$, and a variable set $x$, the least fixed point returns a function $LFP(x, T, S_0)$ where $LFP$ is computed as follows.

   **LFP**$(x, T, S_n)$
   $S_{n+1} = \exists x(T \cap S_n) + S_n$
   **if** $(S_{n+1} = S_n)$
   **return** $S_n$
   **else**
   **return** $LFP(x, T, S_{n+1})$

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

   **GFP**$(x, T, S_n)$
   $S_{n+1} = \exists x(T \cap S_n)$
   **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 by 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 which can reach $A$. It can be computed as follows:

**BR**$(T, A)$

**return** $LFP(x, T(y, x), A(x))$

5. **Reach Reachable Edges Operator**: Given a Transition Relation $T(x, y)$ and a set of edges $E$, the Reach Reachable Edges operator returns the set of vertices which can reach $E$ or be reached by $E$. The *Reach Reachable Edges operator* or $RRE(T, E)$ is computed by the following algorithm:

**RRE**$(T, E)$

$temp1 = \exists_y E(x, y)$

$temp2 = \exists_x E(x, y)$

**return** $(BR(T, temp1) + FR(T, temp2))$

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

**RRS**$(T, E)$

**return** $(BR(T, E) + FR(T, E))$

7. **Cycles** : The set of states in A that are involved in some cycle ( not necessarily contained within A) are computed as

**Cycles**$(T, A)$

$R = FR(T, A)$

$R' = BR(T, A)$

**return** $R \cap R'$

8. **Forward Stable Set Operator**[3]: 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** $(GFP(y, T, A))$

9. **Forward Bad Path Operator**[3]: Given $T(x, y)$, the transition relation, $A(x)$, a set of states and $C$, a set of fairness constraints, the *Forward Bad Path Operator* or $FBP(T, C, A)$, returns a subset of states in $A(x)$ which are on a path satisfying the constraints $C$. 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 $E_i(x, y)$. Note that this operator returns just a path and not necessarily an infinite path. The Forward Bad Path operator FBP can be computed by using the following algorithm:

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

# 3 Previous Work in Language Containment

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 [9]. This 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 $D = M \cap \overline{P}$ is empty.

When $P$ is expressed as an L-automaton, the problem of complementing $P$ is solved by expressing it as an L-process [7]. The acceptance condition 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 has been done implicitly 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 Automaton are the complements of the acceptance conditions for the Streett Automaton. Our experiments are going to be run in the framework of Streett and Rabin Automata. All successive discussions in this report, though centered around Streett and Rabin automata, are also applicable to other classes of automata.

A Language Emptiness check remains at this point and it is performed by checking the product automata, $D$ for acceptable infinite behavior[7], which indicates that the language for the system-property product machine is not empty. A cycle is an integral part of any infinite 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 automata. The set of states which lie on such cycles form a set of "*bad states*" or states which cause the *bad 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 *bad cycle*.

In [3] an algorithm for computation of $Fair^{+}$ are given for Streett automatons. The algorithm computes $Fair^{+}$

by starting with the set of reachable states and alternately applying the FSS and the FBP operators. These operators successively restrict the original set of reachable states to those, which are on a path from an initial state to a cycle (FSS) and those which are on a fair (or acceptable) path (FBP). Thus, the set $Fair^+$ is obtained by successively shrinking the set of reachable states until only those states which are on a fair path from some initial state to a bad (fair) cycle remain. The algorithm for verification in the Streett-Rabin environment becomes:

**Algorithm3.1: Non_Incremental_Language_Containment**

$Fair^+$ **= Compute_Fair**$^+$

**if** $Fair^+$ **is empty return(PASS)**

**else return(FAIL)**

The set $Fair^+$ is computed using the algorithm3.2:

**Algorithm 3.2: Compute_Fair**$^+$

**Restrict the Transition Relation** $T(x, y)$ **to the set of reachable states**

**Remove negative fair edges**

**Set** $S_0$ **= Reachable states**

**While** $S_{n+1} \neq S_n$

$S' = FSS(T, S_n)$

$S_{n+1} = FBP(T, C, S')$

**return** $S_n$

The proof of correctness of this algorithm can be found in [3].

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.

# 4 Incremental Language Containment

## 4.1 Incremental Algorithms

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 successively reduces this set by eliminating states which may not be part of it. Once a state has been removed from this set, it never is added in again and hence the algorithm is monotonic. Our incremental algorithm exploits this observation and is based on the fact that if we supply any superset of $Fair^+$ to the algorithm 3, 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 much faster and hence the incremental algorithm is faster.

## 4.2 Characterizing Incremental Changes

Recall that $Fair^+$ is a set of states that characterize the bad 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; 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 induced in 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 addition or subtraction of edges to the transition relation, addition or subtraction of states ( and hence edges ) to the state space of the machine and addition or subtraction of fairness constraints. Addition and subtraction of states can be characterized in terms of addition and subtraction 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 $T^{new}$. We begin by separating the augmented transition relation $T^{new}$ into $T^{sub}$ and $T^{add}$. $T^{sub}$ consists of the original transitions relation $T$ minus all transitions which were removed in $T^{new}$ and $T^{add}$ is $T^{sub}$ plus all the transitions which are 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 changes in input is described in section 5.

## 4.3 Subtraction of Edges

First, consider the system obtained after deleting a set of edges from the transition relation. Deleting an edge cannot make any unreachable state reachable, nor can it create a new cycle in the state transition graph. Thus, deleting an edge can never add a new state to $Fair^+$. Figure 4 indicates that deleting edge $ab$ can potentially remove all states

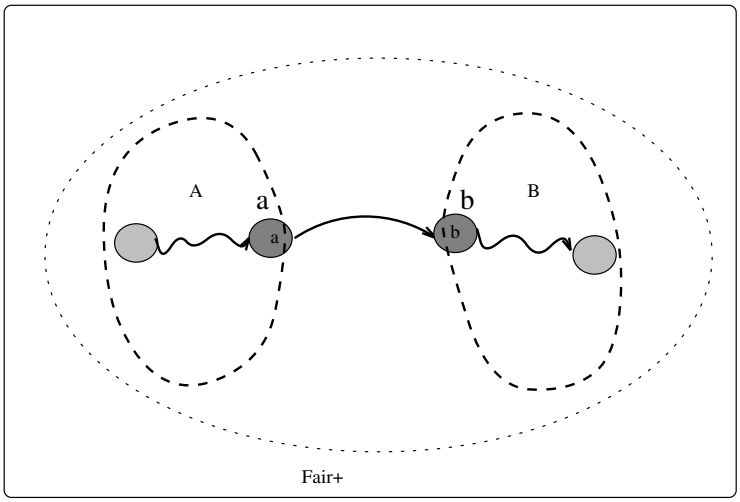


Figure 4: Deleting edge $ab$ can potentially remove all states in $A$ and $B$ from $fair^+$

in sets $A$ and $B$ from the set $Fair^+$. The following lemma formalizes this idea.

**Lemma 4.1** *The set $Fair^{+new}$ obtained 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 removal 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

$R^{sub} = FR(T^{sub}, InitSts)$

**Restrict to new Reachable set** $R^{sub}$

**Remove Negative Fair Edges**

$Fair_0 = Fair^+ \cap R^{sub}$

**While** $Fair_{n+1} \neq Fair_n$

$A = FSS(T^{sub}, Fair_n)$

$Fair_{n+1} = FBP(T^{sub}, A, C)$

**return** $Fair_n$

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

**Proof** From lemma 4.1 and [3] ■

## 4.4 Addition of Edges

Consider the addition of a set of edges to the state transition graph. Addition of edges 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 5 indicates that adding edge $ab$ can

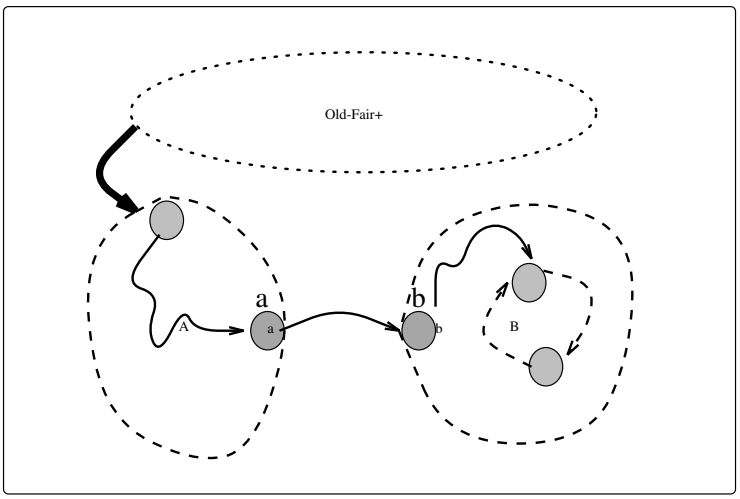


Figure 5: Adding edge $ab$ can potentially add all states in $A$ and $B$ to $Fair^+$

potentially add all states in sets $A$ and $B$ to the set $Fair^+$. However, we will be prove that if the addition of edges results in the addition of one or more states into $Fair^+$ then these states must satisfy at least one of the following conditions in the new Transition system $T^{add}$:

- 1: The state belongs to the set which can reach or be reached by $Fair^+$. This set $Fair^1$ is computed as: $Fair^1 = RRS(T^{add}, Fair^+)$ (Section 2).
- 2: The state can reach or be reached by one of the new transitions. This set $Fair^2$ is computed as: $Fair^2 = RRE((T^{add} \cap \overline{T}), Fair^+)$ (Section 2).

**Lemma 4.3** *The new set $Fair^{+new}$ is a subset of $Fair^1 \cup Fair^2$, i.e. $Fair^{+new} \subseteq Fair^{++} \equiv Fair^1 \cup Fair^2$.*

**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 $s \in Fair^1$, 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 $s \in Fair^2$. This contradicts $s \notin Fair^{++}$. ■

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

**Algorithm 4.4**($T, T^{add}, C, Fair^+$)
$Fair^1 = RRS(T^{add}, Fair^+)$
$Fair^2 = RRE(T^{add}, T^{add} \cap \overline{T})$
$Fair^{++} = Fair^1 \cup Fair^2$
**Remove Negative Fair Edges from** $T^{add}$
$Fair_0 = Fair^{++}$
**While** $Fair_{n+1} \neq Fair_n$
  $A = FSS(T^{add}, Fair_n)$
  $Fair_{n+1} = FBP(T^{add}, A, C)$
**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 [3]. ■

As noted in section 3 the set of reachable states can be used to simplify the BDD for the transition relation. In the previous algorithm the set of reachable states are not explicitly involved but they are 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.

## 4.5 Addition of fairness constraints

The set $Fair^+$ satisfies all the fairness constraints. If new fairness constraints are added to the previous set of fairness constraints, then the new set $Fair^{+new}$ must satisfy all of the older constraints as well as the new ones. The the new 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 $s \in Fair^{+new}$. ∎

The constraints never affect reachability; they only affect the FBP operator (Section 2). Using the new fairness constraints $c$, we can compute a new FBP operator $FBP^{new} = FBP^{old} \cap FBP^c$. The complete computation for the new operator can be written as:

$$FBP^{new}(T, C^{new}, A) = FBP(T, C, A) \cap \prod_{i, c_i \in C^{new}, c_i \notin C} FBP(T, c_i, A);$$

$$FBP^{new}(T, C^{new}, A) = ((\prod_{i, c_i \in C^{new}} (GFP(x, T, A \cdot U_i)) + V_i) \cdot (\prod_{j, E_j \in C^{new}} (GFP(x, T, E_j))));$$

where $U_i, V_i, E_j$ are defined for the computation of the FBP operator, in section 2.

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**

**Compute** $FBP^{new}$

$Fair_0 = Fair^+$

**While** $Fair_{n+1} \neq Fair_n$

$A = FSS(T, Fair_n)$

$Fair_{n+1} = FBP^{new}(T, A, C^{new})$

**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 [3]. ■

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 modified FBP operator to compute the new set $Fair^{new}$. If edges are added then algorithm 4.4 can be used in conjunction with the new FBP 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 new FBP operator (as defined in lemma 4.5) is used in the computation.

**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 FBP operator (as defined in lemma 4.5) is used for the computation.

This section has presented individual algorithms for handling changes in edges and constraints. 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 will describe an incremental algorithm for language containment, when a general change consisting of deletion and addition of edges from the transition structure and addition 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 edges are only added to the transition structure obtained from the previous stage (with the new constraints already in place). Hence, the computation of $Fair^{+new}$ can be decoupled into two stages. 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 transition relation of the processes added, $T^{old}$ 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:

$$T = T^{old} \cap R_1$$

$$T^{sub} = T \cap \overline{T^{new}}$$
$$T^{add} = T^{new} = T \cap T_1$$

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 storing their transition relations and constraints. In our system, the designer is allowed to directly change the individual transition relations and input process constraints, new processes by using the intermediate 'Pif' [4] format.

## 5.2 Incremental Language Containment

The general algorithm for computation of the new $Fair^+$ is based on 4.3 and 4.4, with the additional caveat that the $FBP$ is modified to include the additional 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 $C^{new}$ be the new complete set of constraints. The general incremental language containment (ILC) algorithm becomes:

**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**$^+(T^{add}, T^{sub}, C^{new}, Fair^+)$
$A = Algorithm4.3(T^{sub}, Fair^+, C^{new})$
$Fair^{+new} = Algorithm4.4(T^{sub}, T^{add}, C^{new}, A)$
**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 bad 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 4.4 is valid and the correct set $Fair^{new}$ is returned ( refer to theorem 4.4 and lemma 4.8) ■

# 6 Results

We have implemented the algorithms described in the previous section and tested these on a set of verification benchmarks. Table 1 compares the times taken by the incremental algorithm 5.2.2 (ILC) (described in section 5) to the time taken by the non-incremental algorithm (NLC) (described in section 3). The examples were taken from a set of verification benchmarks in the UC Berkeley HSIS environment and the first column of the table gives the name of the example. The second column describes the time for a single pass of the non-incremental (NLC)algorithm (including the time for non-incremental input update). In the event that the input to the program is not updated incrementally, this column reflects the time taken by the non-incremental algorithm. The last two columns report the times taken by the incremental(ILC) and non-incremental (NLC) algorithms for a general change to the system, which consists of addition, subtraction of edges and addition of constraints. The actual edges that are added or subtracted from the transition relation are arbitrary and are usually chosen so as to make the system pass the language containment check.

| **Example** | **Time NLC[1](s) New Input** | **Time ILC[2](s) Incremental Input Update** | **Time NLC[3](s) Incremental Input Update** |
|---|---|---|---|
| abp | 5.734 | 0.016 | 0.063 |
| counter | 0.222 | 0.257 | 0.289 |
| crd | 1.035 | 0.105 | 0.114 |
| ctlp3 | 1.058 | 0.078 | 0.18 |
| gigamax | 7.491 | 0.02 | 1.382 |
| ping_pong | 0.214 | 0.001 | 0.016 |
| ping_pong_new | 0.77 | 0.008 | 0.035 |
| scheduler | 14.057 | 0.063 | 0.355 |
| short | 0.152 | 0.004 | 0.012 |

Table 1: Comparison of Times for the execution of Non-Incremental Vs Incremental Language Containment Algorithms for a General Change

1: Non-incremental algorithm and non-incremental data input
2: Incremental algorithm and incremental data input
3: Non-incremental algorithm and incremental data input

Since non-incremental algorithms do not account for incremental update of the current input but have to read in the new input( from the beginning), comparing the results for a general non-incremental language containment algorithm (Column 2, Table 1) to the incremental algorithm would be appropriate. However, in order to compare the differences in just the algorithms, the times for the language containment routines, columns 3 & 4 describe the times

for the two algorithms when new input data is always read in incrementally.

While examining these results it is important to recall that these results only correspond to one pass of the incremental algorithm. In practice, many successive passes of the incremental algorithm can be made. The gains for each pass add up and hence, the actual saving is considerably larger. The results show that the incremental algorithm consistently performs better 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 small changes in the input problem. We are currently extending this theory to account for changes that consist of constraint removal. In addition, we are examining alternate methods for inputting changes to the system. We are also searching for a set of larger benchmarks to run our algorithms on, so as to get more conclusive evidence as to 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.

# References

[1] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[2] E. M. Clarke, J. R. Burch, O. Grümberg, D. E. Long, and K. L. McMillan. Automatic Verification of Sequential Circuit Designs. *Phil. Trans. of the Royal Society of London*, 339:105–120, 1992.

[3] 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.*, pages 475–481, Dallas, Texas, June 1993.

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

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

[6] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier Science, 1990.

[7] 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, January 1990.

[8] 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*, pages 130–133, November 1990.

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