Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment

Ramin Hojati[*], Vigyan Singhal[†], and Robert K. Brayton

Department of Electrical Engineering and Computer Sciences

University of California, Berkeley, CA 94720, USA

Abstract

We present the edge-Streett/edge-Rabin environment for doing verification using language containment. This environment has a number of desirable properties compared with the L-process/L-automaton environment ([Kur87b]), which is a practical language-containment-based formal verification environment:

- It contains the L-environment as a subset.
- It can be exponentially more compact than the L-environment.
- We present BDD-based algorithms for main verification functions in this environment, and argue that they are efficient. Furthermore, if the specifications come from the L-environment, our algorithms reduce to the algorithms of [HTKB92] and [HBK93] for the L-environment.
- It is in some sense maximal, i.e. language containment check for the next natural extension to our environment is NP-complete (as opposed to polynomial).

We have implemented our algorithms in our verification tool, and will present a flexible user interface to this environment.

1 Introduction

Automatic formal design verification is the process of raising one's confidence in a design by proving properties of a design. The system is usually modeled using a set of interacting finite-state machines. The semantics of the interaction between the FSM's are given by a concurrency scheme. Two such schemes are interleaving

[*]Research supported by Semiconductor Research Corporation Grant 93-DC-008

[†]Research supported by NSF/DARPA Grant MIP-8719546

and synchronous concurrency models. Regardless of what modeling scheme is used, one ends up with one state machine, which can produce the same set of traces as the original model. Therefore, for our purposes it suffices to assume we are only dealing with one finite-state machine, which we call the product machine.

Since the product machine corresponding to a hardware system can become rather large, abstraction is used to trim some "un-interesting" details. Since abstraction enlarges the set of behaviors of the system, in some cases, it becomes necessary to get rid of some unwanted behavior introduced by abstraction. An example is the case of indefinite pause. Assume we have modeled that a system can pause at a state for an arbitrary but finite amount of time. In this case, we rule out the behavior where the system stays at this state forever by placing a constraint on the model. We refer to such constraints as fairness constraints.

One can use the acceptance condition of (finite) automata on infinite strings as a means of specifying fairness constraints. For example, the acceptance condition of Büchi automata is that an infinite path through the automaton (referred to as a run) is acceptable if it visits one of the final states infinitely often ([Tho90]). To specify that the system is not allowed to stay at state s_0 forever, we mark all other states except s_0 as final states.

Having specified the hardware, which consists of the product machine and the fairness constraints, one proceeds to prove properties of the design. There are two important ways of specifying properties: ω–automata (automata on infinite strings) and Computation Tree Logic (CTL) ([CES86]). In this paper, we are concerned with the ω–automata.

Using language containment for formal verification, one represents a super-set of the desirable behaviors of the system using an ω–automaton T. One can think of T as specifying acceptable patterns for the traces of the system. An example is: only those traces are acceptable where each request is followed by a service. This property can be specified by a two-state automaton. One then checks that the language of the system, which is represented by an ω–automaton S, is included in the language of the property automaton T. Doing so, the user is guaranteed that the system is incapable of producing traces which do not have the desired pattern. This is called verifying a system with respect to a property. The user then continues verifying the system with respect to other properties, until the user is convinced that the intersection of the languages of the properties is equal (or maybe very close) to the desired set of behaviors.

The above scheme for doing formal verification first appeared in literature in [VW86]. [VW86] suggested specifying both the system and the property using Büchi automata, and went on to give an algorithm for language containment in this environment.

The usual method to verify that $\mathcal{L}(S)$ is contained in $\mathcal{L}(T)$, where $\mathcal{L}(S)$ and $\mathcal{L}(T)$ are the languages of the system and property respectively, is by checking that $\mathcal{L}(S) \cap \overline{\mathcal{L}(T)}$ is empty (known as language emptiness check). However, complementing an ω–automaton is a PSPACE-complete problem, and the best known algorithms have exponential complexity ([SVW87], [Saf89]).

The contribution of [Kur87b] was to introduce an environment, where the acceptance conditions of the system and the property are complementary. The system is modeled by an L-process, whereas the property is modeled by a deterministic L-automaton. We refer to this method for verification as the L-environment. Computing the complement of the language of a property is trivial: just think of the L-automaton as an L-process. Based on this paradigm, the first software tool for automatic verification of finite-state systems using language containment was built ([HK90]). The advent of BDD's allows for handling of very large state spaces. [HTKB92] and [HBK93] gave various efficient BDD-based algorithms for language containment and debugging in the L-environment.

In this paper, we introduce a new language containment based formal verification environment. The system is modeled using what we call edge-Streett automata, whereas the properties are modeled using edge-Rabin automata. These automata are generalizations of Streett automata ([Str82]) and Rabin automata ([Rab72]), which are used to specify languages over infinite strings. In terms of the ability to express specifications, this environment (called the RS-environment) is equally expressive as L-environment. However, the RS-environment offers more compactness in expressing specifications. This allows a user to be able to specify the system and properties more succinctly. More importantly, since the equivalent L-environment specification can be exponential in the worst case, the language containment should be much faster in the RS-environment than in the L-environment because of exponentially smaller input size (the algorithms for both L-environment and RS-environment are polynomial in terms of the input size).

Edge-Streett and edge-Rabin automata have complementary acceptance conditions. Hence, the complementation task remains trivial for deterministic edge-Rabin automata. We present efficient BDD-based algorithms for language containment and debugging in the RS-environment which are extensions of the algorithms for the L-environment presented in [HTKB92] and [HBK93]. The RS-environment has several desirable features:

- It contains the L-environment as a subset. Moreover, all algorithms are designed in a manner, so that if the specifications are from the L-environment, the algorithms reduce to their counter-parts in the L-environment. In other words, if we don't use the more expressive features, we don't pay for them.

- There are specifications in the RS-environment, which can be expressed in the L-environment only with an exponential blowup.

- All of the algorithms in the RS-environment are efficient, i.e. running times are expected to be comparable with their counterparts in the L-environment (besides language containment, we present algorithms for early failure detection, debugging, and hierarchical verification). In other words, if the more powerful constructs are used, the price is small.

- We argue that the RS-environment is maximal in the following sense. Language containment in it can be done in (small) polynomial time. We show that the next natural extension causes the language containment problem to become NP-complete.

We have implemented most of our algorithms in our verification framework, currently under development. A more convenient user interface to the RS-environment, which supports various specification means, is described in this paper. It remains to be seen how effectively users are able to express their specifications succinctly using the RS-environment when the same specifications cannot be described equally succinctly in the L-environment.

The flow of the paper is as follows. In section 2, definitions and preliminaries are presented. In section 3, the RS-environment is described, and some theoretical results about its expressiveness are proved. In section 4, the algorithms for various verification tasks are given. Section 5 describes a user interface to the environment. Section 6 concludes the paper.

2 Preliminaries

Definition 1 *A* **Finite State Machine or Automaton** *is a tuple (A, Q, T, I), where A is a finite alphabet, Q is a finite set of states, T is a transition relation on $Q \times A \times Q$, and I is a set of initial states. A* **run** *r of an ω-string x is an infinite sequence of states, such that the first state $r_0 \in I$, and $\forall i$:, $(r_i, x_i, r_{i+1}) \in T$. The set of states occurring infinitely often in a run r, called the* **infinitary set***, is denoted by $\inf(r)$. Since the set of states Q is finite, $\inf(r)$ is always a non-empty finite set.*

Definition 2 *An* **L-process** *is a FSM plus a set of acceptance conditions. The* **acceptance conditions** *are a set of edges R known as* **recur edges***, or a set C of subsets of states known as* **cycle sets***. A string is* **accepted** *if it has an accepting run. A run r is* **accepting** *if no recur edge is traversed infinitely often, and $\inf(r)$ is not contained in any of the cycle sets (i.e. the run does not get stuck in one of the cycle sets forever).*

Remark We use the phrase "acceptance conditions" and "fairness constraints" inter-changeably, especially when we talk about automata used for specifying the system, namely L-processes and edge-Streett automata.

Definition 3 *An* **L-automaton** *is the same as an L-process, except that the acceptance conditions are interpreted in exactly the opposite way: a run r is accepting if at least one of the recur edges is traversed infinitely often, or $\inf(r)$ is contained in one of the cycle sets.*

3 Edge-Rabin and Edge-Streett Automata

3.1 Definitions

Definition 4 *An* **edge-Streett automaton** *S is an automaton A augmented with a set of acceptance or fairness constraints of the following form:*

- *Positive edge constraints. This is a set, each member of which is a set of edges of A.*
- *Negative edge constraints. This is a set of edges of A.*
- *Canonical Fairness Constraints (CFC's). These are a set of constraints of the form $F^\infty(S_i)+G^\infty(T_i)$, where S_i and T_i are sets of states of A, and $1 \leq i \leq h$.*

An ω-string x of A is accepted ($x \in \mathcal{L}(S)$) if there is a run r of x in A such that all of the following conditions are satisfied:

1. *For each set of positive edges, r traverses at least one of the edges in set infinitely often.*
2. *For each negative edge e, r does not traverse e infinitely often.*
3. *For each CFC, either $\inf(r) \cap S_i \neq \phi$, or $\inf(r) \subseteq T_i$.*

Remark CFC's can express a form of fairness known as strong fairness. Hence, CFC's are sometimes referred to as strong fairness constraints.

Definition 5 *An* **edge-Rabin automaton** *R is an automaton A augmented with a set of acceptance or fairness constraints of the following form:*

- *Positive edge constraints. This is a set of edges of A.*
- *Negative edge constraints. This is a set, each member of which is a set of edges of A.*
- *CFC's. These are a set of constraints of the form $F^\infty(S_i) \wedge G^\infty(T_i)$, where S_i and T_i are sets of states of A, and $1 \leq i \leq h$.*

An ω-string x of A is accepted ($x \in \mathcal{L}(R)$) if there is a run r of x in A such that at least one of the following conditions is satisfied:

1. *For positive edges, r follows at least one of the edges in the set infinitely often.*
2. *There is a set of negative edges, such that r does not follow any of the edges in the set infinitely often.*
3. *For at least one of the CFC's, say the $i^{\underline{th}}$ one, $\inf(r) \cap S_i \neq \phi$, and $\inf(r) \subseteq T_i$.*

Notation For an edge-Streett or an edge-Rabin Automaton, a (fairness) constraint is one of the following: a set of negative edges, a set of positive edges, a CFC of the form $F^\infty(S_i) + G^\infty(T_i)$ or $F^\infty(S_i) \wedge G^\infty(T_i)$.

Remark One can translate the edge-Streett (edge-Rabin) automata into Streett (Rabin) automata, which are automata with all fairness constraints expressed as CFC's, with a factor of 2 blow-up in the state space size. The transformation is known as the node-recur transform, and was first described in [Kur87a].

Theorem 3.1 *For every deterministic edge-Rabin automaton R, there is an edge-Streett automaton S, which accepts the complement of the language of R and has at most the same number of fairness constraints.*

Proof: For the set of positive edges in R, create a set of negative edges in S. For the set of sets of negative edges in R, create a set of sets of positive edges in S. For each $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$, create a constraint $G^{\infty}(\overline{S_i}) + F^{\infty}(\overline{T_i})$. Let $x \in \mathcal{L}(S)$. Then x has a unique run in both S and R. Call it ξ. We will show that ξ is rejected in R. Assume not. Then, ξ satisfies one of the three acceptance conditions in Definition 5. Since a string x has a unique run in both R and S, it is easy to see that the unique run of x in S is accepted iff the unique run of x in R is rejected. ■

Remark When BDD's are used for representing sets of states, computing the complement of an edge-Rabin automaton by the above procedure is trivial, since complementing a BDD can be done in constant time.

One may ask whether it is possible to make the environment even more powerful without sacrificing too much efficiency. A natural extension, one may consider, is having acceptance constraints of the form of a product of sums of an arbitrary number of $F^{\infty}(S_i)$'s and $G^{\infty}(T_i)$'s. First, note that sum of a set of $F^{\infty}(S_i)$'s is just equivalent to $F^{\infty}(\bigcup_i(S_i))$. So, only one F^{∞} suffices. It remains to ask whether allowing more G^{∞}'s will sacrifice the efficiency. [EL85] basically answered this question (they were looking at a slightly different problem).

Theorem 3.2 *The problem of determining whether the language of an automaton with acceptance conditions of the form $\prod_i(G^{\infty}(S_i) + G^{\infty}(T_i))$ is NP-complete. is empty,*

Proof: The problem is in NP. The answer is a set of states which can be traversed infinitely often, such that all fairness constraints are satisfied. Just guess this set of states, and check that this set is contained in S_i or T_i for all i. To prove the problem is NP-complete, just use the reduction of [EL85] to show the fair state problem with fairness constraints of the above type is NP-complete. [EL85] reduces 3SAT to a graph with fairness constraints of above type. Now, the language of the automaton is non-empty iff the original formula was satisfiable. ■

Note that the language emptiness check for Streett automata is done in (small) polynomial-time. Hence, the next natural extension to the RS-environment makes the language emptiness check, which is the most important operation in formal verification, NP-complete. This gives the RS-environment a sense of maximality.

3.2 Comparison with the L-environment

In this section, we demonstrate why the RS-environment is more "powerful" than the L-environment. Our conclusion is that in the worst case, L-processes are exponentially less compact than Streett automata. We then present an example of this exponential blow-up for the translation of Rabin automata into L-automata.

3.2.1 Edge-Street automata versus L-processes

In this section we show that there are systems which can be represented compactly as Streett automata but any L-process representation incurs an exponential blowup. Conversely we show that edge-Streett automata contain L-processes, i.e. all L-process specifications can be thought of as edge-Streett automata without any increase in the specification size.

The proof for the following theorem showing compactness of edge-Streett automata relative to L-processes is similar to a proof for demonstrating exponential blowups in conversion of deterministic Streett automata to non-deterministic Büchi automata ([Saf89]).

Theorem 3.3 *For every $n > 0$, there exists a deterministic edge-Streett automaton with $3n$ states and $2n$ accepting pairs such that any equivalent non-deterministic L-process has at least 2^n states.*

Proof: Let the alphabet $\Sigma = \{0, 1, 2\}$. We will look at a word $x \in \Sigma^\omega$ as an infinite sequence of vectors of length n. Let the language $\mathcal{L}$ be the set of all words in which, for all $i \in \{1, \ldots, n\}$, 1 appears in the i-th place in infinitely many vectors iff 2 appears in the i-th place in infinitely many vectors. The language $\mathcal{L}$ is accepted by a deterministic edge-Streett automaton, $S = (\Sigma, Q, \delta, \{q_0\})$ with $2n$ CFC's: $F^\infty(S_i) + G^\infty(T_i)$, where $1 \leq i \leq 2n$. S consists of $3n$ states, $Q = \{q_0\} \bigcup \{(q_i, a) | i \in \{1, \ldots, n\}, a \in \Sigma\}$. For the next state, whenever the automaton reads a letter it jumps to the state with the next higher q_i and a equal to the input symbol. Formally, $\delta((q_i, a), b) = (q_{(i+1) \bmod n}, b)$, and $\delta(q_0, b) = (q_1, b)$. The acceptance condition for S makes sure that for each i, either both $(q_i, 1)$ and $(q_i, 2)$ are included in the infinitary set or neither of them is. Formally, for $i \in \{1, \ldots, n\}$, $S_i = \{(q_i, 1)\}$ and $T_i = Q \backslash \{(q_i, 2)\}$, and $S_{i+n} = \{(q_i, 2)\}$ and $T_{i+n} = Q \backslash \{(q_i, 1)\}$.

Assume that there is a non-deterministic L-process L with less than 2^n states that accepts $\mathcal{L}$. For each $R \subseteq \{1, \ldots, n\}$, let $x_R \in (0+1)^n$ and $y_R \in (0+2)^n$ be vectors such that, $\forall i \in R : x_{R,i} = 1, y_{R,i} = 2$, and $\forall i \notin R : x_{R,i} = y_{R,i} = 0$, where $x_{R,i}$ and $y_{R,i}$ denote the i-th letters of x_R and y_R, respectively. Now the word $z_R = ((x_R)^{2^n} y_R)^\omega$ belongs to $\mathcal{L}$. Consider an accepting run ξ_R and consider equal segments of length $(n2^n + n)$ (call each such segment a *period*) in this run. Each period corresponds to a run over input $(x_R)^{2^n} y_R$. For each period, look at the sequence of 2^n states that occur just before each of the x_R vectors. Since L has less than 2^n states, there exist a state q_R, a non-empty run-segment π_R, and $i, j \in \{1, \ldots, , 2^n\}$, such that for infinitely many periods, ξ_R is in state q_R just before the i-th and j-th x_R vector and π_R is the path segment between these two visits to q_R. Since π_R is traversed infinitely often, none of the edges in this path is a recur edge. Now, since L has fewer than 2^n states, there exist $R \neq R'$ such that $q_R = q_{R'}$. Without loss of generality, $R' \not\subseteq R$. Now we alter the run ξ_R so that whenever we reach q_R, we add the additional path segment $\pi_{R'}$. This will result in an altered run ζ corresponding to a new word w in which, for all $i \in R' \backslash R$, 1 appears infinitely often in i-th place in a vector, but 2 does not. Thus $w \notin \mathcal{L}$. Since none of the extra edges traversed by ζ are recur edges, and since ξ_R is an accepting run (i.e. $\inf(\zeta)$ cannot lie inside a cycle set since $\inf(\xi_R)$, which is a subset of $\inf(\zeta)$, does not), w is accepted by L. Hence, the proof by contradiction. ■

Remark [EL85] provides a practical example of when strong fairness constraints are needed.

We now show that edge-Streett automata specifications contain L-process specifications.

Theorem 3.4 *Given a (deterministic) n state L-process with c cycle sets and r recur edges, there exists a (deterministic) n state edge-Streett automaton with c positive edge constraint sets, one negative edge constraint set containing r edges and 0 CFC's.*

Proof: For each cycle set, construct a set of positive edges such that the destination state of each edge is outside the cycle set. Let the set of negative edges be the set of recur edges. The proof now follows from the definition of L-processes and Streett automata. ■

3.2.2 Edge-Rabin Automata versus L-automata

Most of the properties one is interested in proving about a system, are the so called trace properties, which are properties which should hold of every trace (execution) of the system. The ω-regular languages are the largest set of trace properties known, for which there are automatic verification procedures. All such properties can be expressed by ω-automata.

However, language containment is much easier if the property is expressed as a deterministic automata. It is known that deterministic Rabin automata can specify any ω-regular property ([Tho90]). So, in principle, we do not need to use any non-deterministic properties in the RS-environment. For the L-environment the situation is a little bit worse, since there are ω-regular languages which cannot be expressed by a single L-automaton. However, any ω-regular language can be expressed as an intersection of a set of L-automata ([Kur93]). Language containment is then accomplished by checking language containment against each of these L-automata. Hence, in the L-environment, we may need to perform many language containment checks to check a property which can be done with one check in the RS-environment. We will provide an example of this situation later. Note that any deterministic L-automaton can be trivially expressed as an edge-Rabin automaton (using a construction similar to that in the proof for Theorem 3.4.

Given any arbitrary deterministic Rabin automaton R, the only known procedure to convert that to a set of deterministic L-automata, is to convert R to an equivalent deterministic Muller automaton M and then obtain a set of deterministic L-automata, the cardinality of the set being the number of cycles sets in M ([Kur93]). However, it has been shown in [Saf89] that conversion of a deterministic Rabin automaton to a deterministic Muller automaton incurs a blow-up, the lower bound of which is exponential.

In general, given a deterministic Rabin automaton R, we can also use the following algorithm to get a set of deterministic L-automata, the intersection of whose languages is the same as $\mathcal{L}(R)$.

1. The acceptance condition of R can be thought of a sum-of-products term. Rewrite this term as a product-of-sums (POS), taking the infinitary operators as variables. Note that $\overline{F^\infty}$ = G^∞, and $\overline{G^\infty}$ = F^∞.
2. For each sum term, create an L-automaton, by translating $F^\infty(S_i)$ into a set of recur edges, whose starting points are in S_i, and $G^\infty(T_i)$ into a cycle set $\overline{T_i}$.

The procedure is exponential in the worst case. This procedure does not alter S_i's and T_i's or the transition relation of the automaton. It may be possible that by changing these sets or the transition relation, one can obtain better algorithms, although this is unlikely. To illustrate this point, consider the following example.

Assume that we are given an automaton with $2n$ states, on a complete graph. Assume we have n acceptance conditions of the form $F^\infty(\{2k\}) \wedge G^\infty(\overline{\{2k+1\}})$, i.e. for some k, we visit state $2k$ infinitely often, and we don't visit state $2k+1$ infinitely often. This condition can be expressed as $C = x_1y_1+x_2y_2+\ldots+x_ny_n$, where each x_i is an $F^\infty(S_i)$ condition and each y_i is a $G^\infty(T_i)$ condition. L-automata can only express conditions of the form $x_1 + x_2 + \ldots + x_m$, where each x_i is either an $F^\infty(S_i)$ or a $G^\infty(T_i)$ condition. To represent an arbitrary expression of $F^\infty(S_i)$ and $G^\infty(T_i)$ conditions, we have to express these in a POS of infinitary conditions. Then the expression is realized by a set of L-automata, each one of which expresses one of clauses in the POS form. All these L-automata have the same transition relation (the complete graph on $2n$ states). The condition C is the Achiles' Heel function, whose minimum POS form has 2^n terms ([BHMSV84]). Thus if we transform our system to a set of L-automata with the same transition relation, we get an exponential number of L-automata. However, it may be possible to come up with another set of L-automata, with differing transition relations, so that we can express the property C with polynomial number of L-automata, each polynomially sized in n. This remains an open question and we state it as follows.

Open Question: Given any arbitrary deterministic Rabin automaton with n states and h CFC's, is it possible to obtain s deterministic L-automata $L_1, L_2, \ldots, L_s$, each with polynomial number of states in n and h and polynomial cycle sets in n and h, such that s is polynomial in n and h and $\mathcal{L}(R) = \bigcap_{i=1}^{s} \mathcal{L}(L_i)$?

4 Computations in Edge-Streett/Edge-Rabin Environment

In this section, we show how to generalize computations of [HTKB92] and [HBK93], which are BDD-based algorithms for the L-environment, to the RS-environment. All algorithms have the very desirable characteristics that if the specifications are from the L-environment, the computations are the same as [HTKB92] and [HBK93]'s. If not, the computations still remain efficient. In other words, if we don't use the extra power, we don't pay for it; if we do, the price is small.

4.1 Language Emptiness for Edge-Streett Automata

This is the main verification computation. Usually, it is used in the following scenario. The system is given in terms of edge-Streett automata. The property is given in terms of a deterministic edge-Rabin automaton. The edge-Rabin automaton can be easily and efficiently complemented into an edge-Streett automaton. Then, the language containment check reduces to checking emptiness of the combined edge-Streett automata.

[HTKB92] presented several algorithms for this task in the L-environment. In this paper, we concentrate on their most efficient algorithm and generalize it to the RS-environment. This algorithm depended on several graph operators. We present generalization of each one of these to the case of edge-Streett automata. We assume that the negative fair edges have been removed, and all sets are restricted to the set of reachable states. The task is reduced to finding a fair path in the automaton, (called a *bad path* in [HTKB92]). By following this path, we obtain a string which is accepted by the automaton. This means the language of the automaton is not empty, and hence can act as a counter-example, which is reported to the user.

1. *The forward stable set operator.* Given a set of states S, this operator computes the set of states in S, which can reach some cycle in S. There is a corresponding *backward stable set operator*, which computes the set of states reached from some cycle. These operators remain unchanged. The fixed point computation for the forward stable set operator is $F(x,S) = \nu(X,S) \cdot \exists y(T(x,y) \wedge X(y))$, where $F(x,S)$ is the set returned by the operator, S is the initial set, X is the set we are recurring on, T denotes the transition relation for the graph after the negative fair edges have been removed, and ν signifies a greatest fixed-point computation.

   The fixed point computation for the *backward stable set operator* is $B(S,y) = \nu(X,S) \cdot \exists x(T(x,y) \wedge X(x))$.

2. *The forward bad path operator.* Assume we are given a set of positive fair edges, and n conditions of the form $G^\infty(T_i) + F^\infty(S_i)$. Given a current set of states S, this operator returns a set of states $x \in S$, such that:

   - For each set of positive edges, there is a path in S, starting at x, and reaching some edge in the set.
   - For each condition $G^\infty(T_i) + F^\infty(S_i)$, either $x \in T_i$, or there is a path in S, starting at x, and reaching a state in F^∞.

   The computation for this operator is $FP(S) = \prod_{i=1}^{c}(R^*(x, S_i \cap S) + T_i) \wedge \prod_{j=1}^{p}(R^*(x, E_j))$, where c is the number of CFC's, S_i and T_i are the sets in the i-th CFC $F^\infty(S_i) + G^\infty(T_i)$, $R^*(x, A)$ represents the set of states that can reach the set A, p is the number of positive edge constraints, E_j represents the j-th set of positive edges, and $FP(S)$ is the result of the forward bad path operator.

The corresponding *backward bad path operator* is similar to the forward bad path operator, except that the direction of the path is the reverse. The computation for the backward bad path operator is $BP(S) = \prod_{i=1}^{c}(R^*(S_i \cap S, y) + T_i) \wedge \prod_{j=1}^{p}(R^*(E_j, y))$, where $R^*(A, y)$ represents the set of states that can be reached from A.

3. *The trim operator.* This operator tries to quickly eliminate the set of states, which cannot contain fair behavior, i.e. they cannot reach a fair path. The operator is defined as follows, given an initial set of states S.

    `Perform the following until no more change:` $\{$

    1. `For each` $G^\infty(T_i) + F^\infty(S_i)$`, if` $\overline{T_i} \cap \overline{S_i}$ `can only reach itself, let` $S = S \cap T_i \cap S_i$.
    2. `For each set of positive edges` $\{(a_i, b_i), \ldots, (a_k, b_k)\}$`, if` $\overline{\sum a_i} + \overline{\sum b_i}$ `can only reach itself, let` $S = S \cap (\sum_i a_i) \cap (\sum_i b_i)$.

    $\}$

The final algorithm of [HTKB92] consisted of an initial check for easy failures, and then some main computation. The main part of the algorithm remains unchanged, i.e.

```
Start with reachable states.
Repeat until convergence:
    1. Apply forward bad-path.
    2. Apply forward stable set.
    3. Apply backward bad-path.
    4. Apply backward stable set.
```

Note that in the above algorithm, our graph operators are thought of as transformations, which chop off portions of the current set. The trim operator can be applied at any point of time, and is usually applied before the main computation.

Definition 6 *Let Fair+- denote the set of states, which can reach some fair cycle, and are reached from some fair cycle. A fair cycle is a (non-simple) cycle, whose traversal satisfies all the fairness constraints. A fair state is any state involved in a fair cycle.*

Theorem 4.1 *The above algorithm computes Fair+-.*

Proof: Let the set returned by the above algorithm be U. We will first show *Fair+-* $\subseteq U$. Let $x \in$ *Fair+-*. Then, x can reach a fair cycle. So, it is not deleted by forward bad-path and stable set operators. Similarly, x is reached by a fair cycle. So, it is not deleted by the backward operators. Since, x is a reachable state, we conclude $x \in$ *Fair+-*. Now, let $x \in U$. We will show $x \in$ *Fair+-*. Assume to the contrary. Further assume that x cannot reach a fair cycle (the case that x is not reached by a fair cycle is symmetric). Then, there are two cases:

Case 1. x cannot reach a cycle, in which case, x is deleted by forward stable set operator.

Case 2. x can only reach non-fair cycles. Consider the subgraph reachable from x. Consider a leaf SCC (one which cannot reach any other SCC) of this graph. Call it C. Since C was not deleted by forward bad-path operator, it must be the case that all positive edge constraints are satisfied, i.e. for each set of positive edges, one of the edges is included in C. Now consider the $i^{\underline{th}}$ CFC. Every state in C is either included in T_i or can reach some S_i. Since, states in C can only reach themselves, it follows that C is either included in T_i or contains some node of S_i. Hence, a cycle γ which goes through all nodes and edges of C satisfies the $i^{\underline{th}}$ fairness constraint. We conclude that γ satisfies all positive edge constraints and all CFC's. Hence, γ is fair. But this is a contradiction to x not being able to reach any fair cycles. ∎

Theorem 4.2 *When the system is described by an L-process and the property by an L-automaton, then above computation reduces to that of [HTKB92].*

Proof: This is immediate by noticing,

1. $T_i = \phi$, for all T_i in $G^\infty(T_i) + F^\infty(S_i)$, and S_i's are the complement of cycle sets.

2. There are no positive edge constraints.

∎

Remark The computational complexity of this algorithm is not much different from [HTKB92]'s. The main difference is in the bad-path operator, which only involves taking an extra OR in each computation with respect to a CFC, or a set of positive edges.

4.2 Early Failure Detection

Early failure detection tries to find easily detectable failures. In our environment, a failure translates into an ω–string accepted by an edge-Streett automaton. Again, assume we are given the automaton with negative fair edges removed. Following the same basic hierarchy as [HTKB92], we define the following cycles:

1. Cycles of the first kind. These are cycles contained either in $\bigcap_i(S_i)$ or in $\bigcap_i(T_i)$, and satisfying the positive edge constraints.

2. Cycles of the second kind. These are cycles intersecting $\bigcap_i(S_i)$, and satisfying the positive edge constraints.

3. Cycles of third kind. Any cycle not of the first two kinds.

To find the cycles of the first kind, run the stable set operator on $\bigcap_i(S_i)$. If the resulting set is not empty, run the main computation on this set with the positive edge constraints. If we find no cycles, we can similarly check for cycles of the first kind for $\bigcap_i(T_i)$.

To find cycles of the second kind, run the main computation on the set of reachable states, with the fairness constraints being as follows:

- Positive edge constraints.

- A constraint of the form $F^{\infty}(P)$, with $P = \bigcap_i(S_i)$. If a non-empty set is returned, we have some fair behavior.

Theorem 4.3 *If there are no cycles of the first or second kind, the main computation can be restricted to* $\overline{\bigcap_i(S_i)} \cap R$, *where* R *is the set of reachable states.*

Proof: We are deleting the states in $\bigcap_i(S_i)$'s from consideration. However, because there are no cycles of first or second kind, there cannot be any fair cycles involving states in $\bigcap_i(S_i)$'s. ∎

Theorem 4.4 *In case our specification is an L-environment specification, early failure detection reduces to that of [HTKB92].*

Proof: Similar to the proof for Theorem 4.2. ∎

4.3 Debugging

In this section, we generalize the techniques of [HBK93] for debugging in the L-environment. The problem is to find a "good" debug trace, i.e a string accepted by an edge-Streett automaton, which is short. A debug trace consists of a path to a fair cycle, and the fair cycle itself. We would like to minimize both the path and the cycle. We assume that a set of states containing some fair state is passed to the debugger. [HBK93] presents techniques which guarantee the path to the fair cycle is minimum if a set containing all fair states is passed to the debugger. If a subset T of fair states is given, then a debug trace is returned, whose path is shortest among all debug traces, whose fair cycles have a representative in the set T. The debugger also heuristically minimizes the length of the fair cycle. In this section, we generalize these techniques. Again, our computation reduces to [HBK93]'s if an L-environment specification is given.

4.3.1 Finding a Path to a Fair Cycle

First, we discuss how the path to a fair cycle, which is minimum (in the above sense) is computed. Given the set T passed to the debugger, we start with the set of initial states, and do a breadth-first search until some state $x \in T$ is encountered. If the strongly-connected component containing x (denoted by $SCC(x)$, and computed as in [HBK93]) is fair, we are done. Otherwise we take out $SCC(x)$ from T and continue the search. The algorithm can be summarized as follows:

```
0.  Let T be the set passed to the debugger.
1.  Let frontier_set = (initial_states ∩ T), cur_reached = frontier_set, cur_T = T.
2.  Until frontier_set ≠ φ {
     Pick a state x in frontier_set.
     If SCC(x) is fair, then done.  If not, delete SCC(x) from frontier_set and cur_T.
}
3.  Set frontier_set and cur_reached to the intersection of cur_T and the set of states
     reachable from cur_reached in one step.  Return to step 2.
```

The check for a fair SCC is more complicated than that of [HBK93], and will be described below. The idea is to check, for each CFC $F^\infty(S_i)+G^\infty(T_i)$, whether the SCC W intersects S_i. If not, we restrict W to T_i and continue until all CFC's can be satisfied. The CFC's which cause such a restriction can be marked as inactive, since further restrictions will not affect their the satisfaction of these CFC's. After all CFC's have been satisfied, we check if all the positive edge constraints can be satisfied. The entire algorithm runs as follows:

```
0.  Let an SCC W be given.
1.  If W is a single state with no self-loops, return FALSE.
2.  Otherwise, let cur_W = W.  Mark all F∞(S_i) + G∞(T_i)'s as active
3.  Do the following for every active F∞(S_i) + G∞(T_i) until cur_W does not change any more:
     if cur_W ∩ S_i = φ {
          Let cur_W = cur_W ∩ T_i, and mark the i^th constraint as inactive.
          If cur_W = φ return FALSE.
     }
4.  If all the positive edge constraints are satisfied (i.e.  for each positive edge
     constraint, some edge in that constraint is present in T(x,y) restricted to cur_W),
     return TRUE. Otherwise, return FALSE.
```

Note that the above algorithm can delete some states in $SCC(x)$, in which there cannot be fair behavior. In this case, we have to compute a closest state in *cur_W* to *cur_reached*. But, there may be a fair cycle now closer than the one found by this procedure. To find it, we have to start expanding from *cur_reached* and examining various SCC's until *cur_W* is reached or a closer fair SCC is found. In practice, we will just take *cur_W* as the fair SCC. In this case, a closest state in *cur_W* to *cur_reached* is computed, which will serve as the initial state of the fair cycle (the computation of the fair cycle is described in the next section).

4.3.2 Finding a Short Fair Cycle in a Fair SCC

The algorithm of section 4.3.1 returns a set of states W containing a fair cycle, and a state x which serves as the starting point in the cycle. It also guarantees that $S_i \cap W \neq \phi$ for all active $F^\infty(S_i) + G^\infty(T_i)$'s. Hence, we can restrict our attention to S_i's of active ones. Now, we need to find a short cycle which visits a state in each S_i. The algorithm of [HBK93] finds a fair cycle, when the fairness constraints are of the form $\prod F^\infty(S_i)$. If we disregard the positive edge constraints, since all inactive CFC's are already satisfied and for all active CFC's $S_i \cap W \neq \phi$, we can call their algorithm on S_i's of the active CFC's. Note that this solution may be sub-optimal since we have not considered the T_i's of the active CFC's. For example, assume W is a complete graph on 2 vertices, and we have a CFC $F^\infty(\{2\}) + G^\infty(\{1\})$, and the starting state is 1. Our algorithm will return the cycle $(1,2)$, whereas the cycle consisting of the self-loop on 1 is a shorter fair cycle.

Also, the positive edge constraints may not be satisfied by the solution returned by the algorithm of [HBK93]. To satisfy a given positive edge constraint (consisting of the set of edges $E_i(x,y)$), if $C = (c_0, \ldots, c_{m-1})$ is the fair cycle returned by [HBK93]'s algorithm, we look for a closest state c_j in C to the starting vertices of the edges in $E_i(x,y)$. Let the chosen starting vertex be u. We then choose a closest path from some vertex v to $c_{(j+1) \bmod m}$, such that $(u,v) \in E_i(x,y)$. We call this operation *patching the cycle* with respect to a positive edge constraint. We repeat this process for all positive edge constraints.

The algorithm to find a fair cycle is, thus, summarized as:

1. `Take all the active` $F^\infty(S_i) + G^\infty(T_i)$`'s.`
2. `Find a cycle by calling algorithms of [HBK93] on` S_i`'s of active` $F^\infty(S_i) + G^\infty(T_i)$`'s.`
3. `Keep patching the cycle with respect to each positive edge constraint until all such constraints are satisfied.`

4.4 Hierarchical Development

We look at the design process as a hierarchical activity. The designer first abstracts the design, and proves some properties about it. At the next stage, the designer adds more detail to the design. For example, a module is implemented by three modules. By definition of a subsystem implementing another, the behavior of the implementation at a lower level must be contained in the behavior of the specification at a higher level. This means that language of the lower level implementation must be contained in the language of the higher level specification.

Since systems are specified using edge-Streett automata, this means that we need to check language containment between two edge-Streett automata. [Kur93] presents an algorithm for language containment of two L-processes (this comes up in the environment described in [Kur93] because of application of homomorphic reductions). We present an algorithm with the same flavor for our problem.

Assume two edge-Streett automata S_1 and S_2 are given. We assume that S_2 is deterministic. Using the following two lemmata, we reduce the problem of containment of $\mathcal{L}(S_1)$ into $\mathcal{L}(S_2)$, into a set of language containments between edge-Streett and edge-Rabin automata.

Lemma 4.5 *$\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$ iff $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_i)$, for $1 \leq i \leq n$, where n is the number of fairness constraints of S_2, and each S_i has the same transition structure of S_2, but only one of the fairness constraints of S_2 (the set of negative edges is considered as one fairness constraint, whereas each positive edge and each constraint $F^\infty(S_i) + G^\infty(T_i)$ is a separate fairness constraint).*

Proof: $(\Rightarrow)$
Assume $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2)$. Let x be a trace of S_1. We have to show x is a trace of each S_i. Assume x is not a possible trace of some S_j. Then, that fairness constraint is not satisfied on x. Hence, x is not a trace of S_2. So, $\mathcal{L}(S_1) \not\subseteq \mathcal{L}(S_2)$, which is a contradiction.

$(\Leftarrow)$
Assume to the contrary that $\mathcal{L}(S_1) \not\subseteq \mathcal{L}(S_2)$. Let x be a string in $\mathcal{L}(S_1)$ and not in $\mathcal{L}(S_2)$. Then, some fairness constraint of S_2 is not satisfied on x. Let it be constraint j. Then, x is not a trace of S_j, which is a contradiction. ∎

Lemma 4.6 *If a deterministic edge-Streett automaton S has only one fairness constraint, then one can build a deterministic edge-Rabin automaton, accepting the same language, which has the same transition structure as S, and at most two fairness constraints.*

Proof: We have to show the transformation for each kind of fairness constraint. We have three cases:

1. The constraint is a negative edge constraint. Create a Rabin automaton with these edges being negative edge constraints.

2. The constraint is a positive edge constraint. Create a Rabin automaton with these edges being positive edge constraints.

3. The constraint is of the form $F^\infty(S_i)+G^\infty(T_i)$. Create a Rabin automaton with two CFC's: $F^\infty(S_i) \wedge G^\infty(Q)$, and $F^\infty(Q) \wedge G^\infty(T_i)$, where Q is the set of all states in S.

It is easy to see that for each of the above three transformations, the language of the deterministic edge-Rabin automaton is the same as the language of the corresponding edge-Streett automaton. ■

Remark Note that the our algorithm for language containment between two edge-Streett automata is rather efficient. For checking whether $\mathcal{L}(S_1)$ is contained in $\mathcal{L}(S_2)$, we have to solve n problems (n being the number of constraints of S_2), each of which can be solved very efficiently. Since, these fairness constraint are given by the user, we expect their number to be small.

Remark One can also use the algorithm reported in [Saf92] for the above task. Given a deterministic Streett automaton with n states and h CFC's, [Saf92]'s algorithm returns a deterministic Rabin automaton with $n2^{h \log h}$ states and $h+1$ CFC's accepting the same language. Note that this algorithm is exponential in the number of CFC's. If this number of small, as it is expected to be in practice, this algorithm may be practical.

If S_2 is non-deterministic, then the procedure is more expensive. [Saf92] proposed an algorithm for determinizing a non-deterministic Streett automaton. This algorithm returns a deterministic Rabin automata. Then, the following algorithm decides whether $\mathcal{L}(S_1)$ is contained in $\mathcal{L}(S_2)$, when S_2 is non-deterministic.

1. `Build` S_2'`, the node-recur transform of` S_2
2. `Determinize` S_2'`, and get a deterministic Rabin automata,` S_2''`.`
3. `Check if` $\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2'')$ `using the techniques specified in section 4.1.`

Theorem 4.7 *If S has n states and h CFC's, then S_2'' in the above algorithm, has at most $2^{n \log n}$ states.*

Proof: This follows from the fact that step 1 increases the state space size by a factor of 2, and the complexity result of [Saf92]. ■

5 User Interface

Given that our final environment is the edge-Streett/edge-Rabin environment, we can create a more flexible user interface. The users specify their processes, using finite state machines. They can attach various kinds of fairness constraints to their FSM's. There are two general categories of fairness constraints: positive and negative. The positive fairness constraints are a set of constraints defining what traces are acceptable, whereas the negative ones define which ones are not.

5.1 Fairness Constraints

There are three types of positive fairness constraints:

1. *Nodes.* A set of nodes one of which has to be traversed infinitely often.

2. *Edges.* A set of edges one of which has to be traversed infinitely often.

3. *CFC.* A constraint of the form $F^{\infty}(S_i) + G^{\infty}(T_i)$.

There are four types of negative fairness constraints:

1. *Node.* A node which can only be visited finitely many times.

2. *Edge.* An edge which can only be visited finitely many times.

3. *Subset.* A set of states which should either be exited infinitely often, or be visited only finitely often.

4. *CFC.* A constraint of the form $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$, which should not be satisfied.

A string is accepted if it has a run where all the positive and negative fairness constraints are satisfied. The above constraints can be attached to each machine or to the whole system. If a constraint is attached to an individual machine, then any satisfying run must satisfy that constraint on that machine. A constraint attached to a single machine has a different meaning when viewed in the product machine environment. For example, a positive fair node of an individual machine translates into a set of positive fair nodes for the product machine.

One can also attach a constraint to a set of machines. For example, if we would like to say a fair node is one where the first machine is at state 3 and the second machine is at 5, we can do so. Note that one does not have to specify what happens to other machines.

Theorem 5.1 *A system with the above fairness constraints can be expressed as an edge-Streett automaton.*

Proof: We show how each non-standard fairness constraint supplied by the user can be converted to one compatible with Definition 4. Mark the incoming and outgoing edges of a set of positive nodes as a set of positive edges. Mark all incoming and outgoing edges of a negative node as negative edges. Transform each negative subset constraint on the set S into $F^{\infty}(\overline{S})+G^{\infty}(\phi)$. Transform each negative CFC $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$ into $G^{\infty}(\overline{S_i}) + F^{\infty}(\overline{T_i})$. ∎

5.2 Acceptance Conditions

For specifying properties, we again have two sets of acceptance conditions (for properties, we choose to call them acceptance conditions): positive and negative acceptance conditions. There are three types of positive acceptance conditions:

1. *Node.* A node which has to be traversed infinitely often.

2. *Edge.* An edge which has to be traversed infinitely often.

3. *CFC*. A constraint of the form $F^\infty(S_i) \wedge G^\infty(T_i)$.

There are four types of negative acceptance constraints:

1. *Nodes*. A set of nodes which can only be visited finitely many times.

2. *Edges*. A set of edges which can only be visited finitely many times.

3. *Subsets*. A set of states which should either be exited infinitely often, or be visited finitely often.

4. *CFC*. A constraint of the form $F^\infty(S_i) + G^\infty(T_i)$, which should not be satisfied.

A string is accepted if one of the positive or negative acceptance constraints is satisfied.

Theorem 5.2 *An automaton with the above acceptance conditions can be expressed as an edge-Rabin automata.*

Proof: The proof of this lemma is similar to the proof of Theorem 5.1. ■

6 Conclusions

In this paper, we have presented a new environment, the RS-environment, for formal verification using language containment. Efficient BDD-based algorithms for this environment were given. The RS-environment contains the L-environment as a subset, although it can in some cases be exponentially more compact. All of the algorithms presented have the nice property that if the specifications are from the L-environment, the algorithms reduce to their counter-parts in the L-environment. If not, the algorithms remain efficient. The algorithms have been implemented in our verification framework, and are ready to be tested in the future.

References

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[EL85] E.A. Emerson and C.L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[HBK93] R. Hojati, R.K. Brayton, and R.P. Kurshan. BDD-Based Debugging of Design Using Language Containment and Fair CTL. In *Proceedings of the Conference on Computer-Aided Verification*, Elounda, Crete, Greece, June 1993. To appear.

[HK90] Z. Har'El and R.P. Kurshan. Software for Analytical Development of Communication Protocols. *AT&T Technical Journal*, pages 45–59, January 1990.

[HTKB92] R. Hojati, H. Touati, R.P. Kurshan, and R.K. Brayton. Efficient ω-Regular Language Containment. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, pages 371–382, Montréal, Québec, Canada, 1992.

[Kur87a] R.P. Kurshan. Complementing Deterministic Büchi Automata in Polynomial Time. *Journal of Computer and System Sciences*, 35:59–71, 1987.

[Kur87b] R.P. Kurshan. Reducibility in Analysis of Coordination. In *Discrete Event Systems: Models and Applications*, volume 103 of *LNCIS*, pages 19–39. Springer Verlag, 1987.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, 1993. To appear.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics.* American Mathematical Society, Providence, Rhode Island, 1972.

[Saf89] Shmuel Safra. *Complexity of Automata on Infinite Objects.* PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, March 1989.

[Saf92] S. Safra. Exponential Determinization for ω-Automata with Strong-Fairness Acceptance Condition. In *Proceedings of the ACM Symposium on the Theory of Computing*, 1992.

[Str82] R.S. Streett. Propositional Dynamic Logic of Looping and Converse is Elementary Decidable. *Information and Control*, 54:121–141, 1982.

[SVW87] A.P. Sistla, M.Y. Vardi, and P.L. Wolper. The Complementation Problem for Büchi Automata, with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Tho90] 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.

[VW86] M.Y. Vardi and P.L. Wolper. An Automata-Theoretic Approach to Program Verification. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 332–334, 1986.