An Environment for Formal Verification Based On Symbolic Computations

Ramin Hojati
Robert K. Brayton
University of California, Berkeley

Abstract

We present a formal verification environment for hardware based on symbolic computations. The hardware is described using our concurrency model, called the combinational/sequential concurrency model, a powerful modeling scheme with close ties to hardware. Because non-determinism is allowed, sometimes unwanted behavior is introduced. To remove this, fairness constraints are used. Having specified the model, the user can then prove properties of the design using either CTL or edge-Rabin automata. In this paper, we present our concurrency model, and algorithms for language containment checking, fair CTL model checking, early failure detection, and hierarchical verification.

1 Introduction

Design Verification is the process of answering the question of "Is what I specified what I wanted?" This is accomplished by specifying the system at a suitable level of abstraction and then proving properties of the system. For example, in a large design one can abstract the design to functional blocks and verify that the communication between blocks is correct. An example of a property may be that no two functional units write to the global bus at the same time, a so-called safety property.

The ***behavior of a system*** is its set of traces. A ***trace*** is a possible execution of the system, i.e. an infinite sequence of outputs of a system. One can view the outputs of a system as a tuple. In this manner, the outputs define an alphabet Σ; a trace is an ω-sequence (infinite sequence) of letters from Σ, such that the first letter is the tuple of the outputs of the system at an initial state, and every state is reachable from the previous one by some set of inputs. The most widely used modeling scheme is a set of interacting non-deterministic finite state automata. This style of specification is supported by our concurrency model. Non-determinism is used heavily for abstraction. For example, in many cases, the environment is modeled as part of the system by a non-deterministic automaton which produces any input at any point in time.

To model (possibly non-deterministic) systems, we use the ***combinational/sequential concurrency model*** (***CS***), which closely resembles synchronous hardware. A system expressed in CS consists of a set of non-deterministic tables (intuitively gates) connected by wires. On some wires, special elements called latches may be placed, which hold values. At every time-step, a subset of latches update their values, and then propagate these values through the tables. In this paper, we are only concerned with synchronous CS, where all latches update their values at every time-step. Synchronous CS is a powerful model, capable of compactly describing various kinds of behavior, including asynchronous behavior.

A system in synchronous CS can be thought of as a set of interacting FSMs, where each latch represents a FSM. One can represent the same behavior using one FSM. However, this FSM may have a large number of states; the ***state explosion problem***. Binary Decision Diagrams (***BDDs***) ([Bry86]) or ***symbolic computations*** are a means of dealing with large state spaces. Our algorithms use BDDs as their main data structure.

Abstraction may result in unwanted behaviors. An example is in modeling unbounded delays, where we want to allow the system to stay at a state s for some finite but unknown amount of time. The behaviors where the system is in s forever should be excluded. An easy abstraction would allow infinite delays, but then all properties of the system should only be checked on traces, where the system gets out of s infinitely often if it enters s infinitely often. A second example of unwanted behavior is in modeling schedulers. Assume that a set of processes are executed in a system controlled by a ***fair scheduler***: one which disallows starvation of processes. An example of a fair scheduler is a round robin scheduler. To avoid having to model the details of a particular scheduler, system constraints can be used to disallow looking at traces where one process is blocked forever. Such constraints on a system's traces are called ***fairness constraints*** (***FCs***). In our environment, abstraction and fairness constraints are allowed. Hence, for us a hardware system consists of a model of the hardware plus a set of fairness constraints. Although we allow the user to specify the fairness constraints in various forms, internally, they are translated into edge-Streett automata (a minor extension of Streett automata [Str82]).

Having specified the system, the next step is the verification of properties. For verification, one can use Computation Tree Logic (CTL) ([Cla86]) formulas, or edge-Rabin automata. To verify a property expressed in CTL, model checking is used, which is the process of verifying a system satisfies a logical formula. Because of presence of fairness constraints, this process is referred to as ***fair CTL model checking*** (***MC***). To verify a property expressed using edge-Rabin automata (a minor extension of Rabin automata [Rab72]), we use ***language containment*** (***LC***), which is the process of verifying the language of a system is contained in the language of a property. Our language containment environment is called the ***edge-Streett/edge-Rabin (eSeR) environment***. Language containment is performed by checking language emptiness, where we check that the intersection of the complement of the language of the property automaton and the language of the system is empty.

We also present algorithms for MC and LC. In both our MC and some of our LC algorithms, we need to compute a set of states, called ***fair reachable states***, denoted $Fair+$, which can reach a ***fair cycle***, one whose traversal satisfies all the fairness constraints. We present several algorithms for computing the fair reachable states, which are based on ideas of [Eme87] on how to translate fair CTL formulas into μ-calculus. There are many ways to compute this set ([Hoj92]), but only one is presented here.

For LC, we describe an alternative algorithm, which views language containment as a sequence of transformations which trim portions of the state space, where no bad behavior can occur. ***Bad behavior*** in the context of LC is a ***fair path***, i.e. a path satisfying all fairness constraints.

To motivate our decision to support both MC and LC, we show that LC and MC are somewhat complementary, i.e. although there is a subset of common properties expressible by either, there are properties which only LC or MC can express. Moreover, some properties are more easily expressed or verified using one of the two methods. Hence, it is useful to support both.

In practice, we expect a verification tool to be used in a debugging environment where more often properties fail than pass. Hence, it is important that the tool be able to quickly find some easy errors. We present techniques for finding easy errors. One involves computing a subset of states, reachable in a few steps. If the property is expressed by edge-Rabin automata or in ***ACTL*** (a subset of CTL where no existential quantification is allowed, and negations are only applied to atomic propositions) finding an error in this subset implies that the error exists in the whole reachable set as well. For LC, we present effective early failure techniques which rely on the structure of the graph induced by fairness constraints.

Hierarchical verification is the process of verifying properties at a higher-level of abstraction, but being ensured that the properties continue to hold at lower levels. The recommended design methodology in our environment is to write specifications in abstract terms, using non-determinism and fairness constraints. A set of properties is then verified at this level. Parts of the design are then refined. If these refinements do not add behavior, then the properties proved using LC at high-levels of abstraction are guaranteed to hold at the lower levels.

Much of the material and ideas presented here has appeared in conference papers, [Hoj92], [Hoj93a], and [Hoj93c]. Most of the algorithms described have been implemented in our verification tool, ***HSIS***, being developed at University of California, Berkeley. An important part of our environment are additional debugging techniques which are not addressed here. These attempt to feed back to the user a succinct error trace which is used to more easily locate a bug. Interested readers are referred to [Hoj93b].

The paper's flow is as follows. Section 2 presents the preliminaries, including the combinational/sequential concurrency model. Section 3 describes how fairness constraints are described. Section 4 presents algorithms for computing fair reachable states. Section 5 and 6, give algorithms for MC and LC respectively. Section 7 argues that having both LC and MC in one environment are useful. Section 8 describes techniques for early failure detection. Section 9 details our hierarchical verification schemes, and presents the algorithms needed for this task. Conclusions are given in section 10.

2 Preliminaries

2.1 Some Definitions

Definitions A ***Finite State Machine*** (***FSM***) or ***Automaton*** is a tuple (Σ, Q, T, I), where Σ 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 in an automaton A is an infinite sequence of states, such that the first state $r_0 \in I$, and for all 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. The set of all ω-strings which have a run in A is called the ***behavior*** or ***language*** of A.

2.2 Combinational/Sequential (CS) Concurrency Model

In this sub-section, we describe the combinational/sequential (CS) concurrency model. There are four primitives: variables, tables (intuitively non-deterministic gates), wires, and latches.

A ***variable*** takes values from some finite domain. A ***table*** is a relation defined over a set of variables. The domain of a table is the cartesian product of the domains of its variables. The variables of a table are partitioned into inputs and outputs. Tables are connected by ***wires***, which connect an output variable of a table to an input variable of a table (where both variables are the same). We assume a variable is the output of at most one table. The possible values of a wire is the set of elements in the domain of its variable. Special elements called ***latches*** may be placed on wires. When a latch is placed on a wire, the wire is broken into two pieces, the input to the latch, and the output of the latch. We assume our system is a closed system, hence the input to every table or latch is the output of some other table or latch. With every latch we associate a set of initial values, which are a subset of the set of values of its wire. A ***state*** is an assignment of values to the latches of a model, where a value assigned to a latch must be in its domain. An ***initial state*** is a state where every latch takes a value from its set of initial values.

If the tables are deterministic, then we can think of them as gates. In this case, the above model is syntactically the same as synchronous hardware. The concurrency model of ***synchronous CS*** is exactly the same as synchronous hardware also, modulo the fact that tables may be non-deterministic. At every point, the system is in some state, with the ouputs of the tables having some stable (consistent) set of values. At every step, all latches update their values, i.e. the outputs of the latches take the values of their inputs. These values then propagate through the tables, until all lines have a consistent set of values (hence, oscillatory behavior is not considered). If during propagation a latch is encountered, i.e. an output of a table is an input to a latch, the propagation of values is stopped. Note that because of non-determinism, given a state, there may be several consistent sets of values. This process continues indefinitely in time. We assume the first state is an initial state of the model.

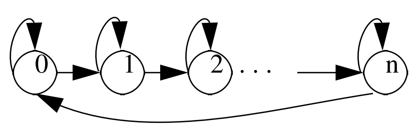
To represent the behavior of the system, we represent each table by a BDD, more strictly an MDD ([Kam92]). For each latch, the input of the latch is the "next state" variable, and the output is the "present state" variable. The product of all the BDDs is a transition relation $T(x, i, y)$ which describes the (stable) behavior of the whole system, where x and y represent the set of present and next states respectively, and i represents the non-state variables. This transition relation is used for simulation and debugging, where information about non-state variables is needed. For verification, we existentially quantify out all non-state variables, and get a transition relation denoted by $T(x, y)$. One can think of this ***product machine*** as one FSM, whose set of states is the Cartesian product of the set of values of all the latches, and whose transitions relation between states is given by $T(x, y)$.

3 Fairness Constraints

To restrict the behavior of abstract FMSs, fairness constraints are introduced. Fairness constraints put restrictions on what runs are ***accepting***. An ω-string is accepted, i.e. is part of the behavior of the system, iff it has an accepting run, where the acceptance condition comes from the fairness constraints. Fairness constraints can be divided into negative and positive constraints. Within each category, various kinds of fairness constraints are allowed.

3.1 Positive Fairness Constraints (PFC)

We motivate positive fairness constraints by an example. Consider the scheduler given below. This schedules processes $0, \ldots, n$ in a round rabin fashion, except that each process may be activated many times. If no FCs are imposed, then the scheduler can schedule one of the processes forever. To impose fairness, we restrict the behavior of this machine, by asserting that only those behaviors are accepting, where the edge $(n, 0)$ is taken infinitely often. Intuitively, this means that the scheduler keeps cycling through all the processes.



This scheduler schedules processes in a round rabin fashion, allowing each process to continue for an arbitrary amount of time. To ensure fairness, the edge (n,0) is declared as a positive fair edge, one which must be taken infinitely often.

Figure 1

There are three types of PFCs.

1. ***Positive Fair Nodes***. A set of nodes one of which must be taken infinitely often (∞-often). Hence, a run is acceptable only if one of the positive fair nodes is visited ∞-often in the run.
2. ***Positive Fair Edges***. A set of edges one of which has to be traversed ∞-often.
3. ***Positive Fair Subset***. A set of states which must be visited ∞-often and exited only finitely many times. Hence, if S is a pos-

itive fair subset, a run r is accepting iff $inf(r) \subseteq S$.

4. ***Positive Canonical Fairness Constraints (PCFC)***. A constraint of the form $F^{\infty}(S_i) + G^{\infty}(T_i)$, where S_i and T_i are subsets of nodes. This constraint says that a run r is acceptable iff $inf(r) \cap S_i \neq \emptyset$ (r visits the set S_i ∞-often) or $inf(r) \subseteq T_i$ (after some point, r consists only of states in T_i).

3.2 Negative Fairness Constraints (NFC)

Similarly, there are four types of NFCs.

1. ***Negative Fair Node***. A node which should be visited only finitely many times, i.e. it should not be traversed ∞-often.
2. ***Negative Fair Edge***. An edge which can be visited only finitely many times.
3. ***Negative Fair Subset***. A set of states which either should be visited finitely many times, or exited ∞-often. Hence, if S is a negative fair subset, a run r is accepting iff $inf(r) \cap \bar{S} \neq \emptyset$.
4. ***Negative Canonical Fairness Constraints (NCFC)***. A constraint of the form $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$, where S_i and T_i are subsets of nodes. This constraint says that a run r is acceptable iff $inf(r) \cap S_i \neq \emptyset$ and $inf(r) \subseteq T_i$.

Note that a run is accepting iff all the positive and negative fairness constraints are satisfied.

3.3 Edge-Streett Automata

To make algorithm design easier, we translate a FSM with the above fairness constrains to a more restricted form, known as edge-Streett automata.

Definition An ***edge-Streett automaton*** is a FSM with the following fairness constraints: a set of negative fair edge constraints, a set of positive fair edges constraints, and a set of positive canonical fairness constraints. A run is accepting if all of the fairness constraints are satisfied.

Theorem A system with the above positive and negative fairness constraints can be expressed as an edge-Streett automaton, so that the set of accepting runs is the same in both systems.

Proof We show how each positive or negative fairness constraint can be converted to one of the fairness constraints allowed by edge-Streett automata. Since, negative fair edges constraints, positive fair edges constraints, and PFCs are allowed by edge-Streett automata, we only need the following transformations.

1. Positive Fair Nodes: Mark the incoming and outgoing edges of a set of positive nodes as a set of positive edges.
2. Negative Fair Node: Mark all incoming and outgoing edges of a negative node as negative edges.
3. Negative Fair Subset: For each negative fair subset S, create a PCFC of the form $F^{\infty}(\bar{S}) + G^{\infty}(\emptyset)$.
4. NCFC: Transform a NCFC $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$ to a PCFC $F^{\infty}(\bar{T_i}) + G^{\infty}(\bar{S_i})$.

It is easy to see that with the above transformations, a run is accepting according to positive and negative fairness constraints iff it is accepting according to edge-Streett fairness conditions (QED).

4 Computing Fair States

At the heart of MC and many LC algorithms, is a function which computes the set of all fair reachable states, i.e. those from which there is an accepting run. An accepting run according to fairness constraints is called a ***fair path***. In this section, we present an algorithm for this task. Our algorithm is a modification of the Emerson-Lei computation, presented in [Eme87], and re-formulated in the language containment environment by [Tou91]. [Hoj92] presented several other ways of computing this set. In what follows, let G be a graph, V the set of its vertices, $A \subseteq V$ a subset of its vertices, and $T(x, y)$ its transition function.

Definition A ***cyclic strongly connected component (CSCC)*** of G is an SCC of G which contains at least one cycle. The remaining are single node SCCs with no self-loop. These are called ***acyclic strongly connected components (ASCCs)***.

Definition Let F be a monotone increasing k-ary predicate transformer. Define the ***least fixpoint of*** F ***given*** Q, denoted by $\mu(X, Q).FX$ where Q is an k-ary over sets $D_1, \ldots, D_k$, by the set $F^i(Q)$ such that $F(F^i(Q)) = F^i(Q)$. Intuitively, X is the variable we are recurring on, and Q is its initial value. Similarly, define the ***greatest fixpoint*** of a monotone decreasing k-ary predicate transformer F given Q, denoted by $\nu(X, Q).FX$, by the set $F^i(Q)$ such that $F(F^i(Q)) = F^i(Q)$.

Definition Let $S_1(A, y) = \exists x(A(x) \wedge T(x, y))$ and $S_1(x, A) = \exists y(A(y) \wedge T(x, y))$. Thus, $S_1(A, y)$ are the successors of $A(x)$ and $S_1(x, A)$ are the predecessors of $A(x)$.

Definition Let $R_1(A, y) = A(y) \vee S_1(A, y)$, $R_1(x, A) = A(x) \vee S_1(x, A)$, $R^*(A, y) = \mu(X, A).(R_1(X, y))$, $R^*(x, A) = \mu(X, A).(R_1(x, X))$. Note that the first two are "one-step" operators, while the last two compute the least fixed-point containing A. One should read $R^*(A, y)$ as the set of points reachable from A. Similarly $R^*(x, A)$ is the set of points that can reach A. $R_1(A, y)$ and $R_1(x, A)$ are the one-step versions of these respectively.

Definition We define two "stable set" operators, $S^*(A, y) = \nu(X, A).(S_1(X, y))$ and $S^*(x, A) = \nu(X, A).(S_1(x, X))$. One can think of the first as calculating the ***backward stable set*** contained in A (i.e. the set of all vertices which are reached by some vertex involved in some cycle in A), and the latter as the ***forward stable set*** of A (i.e. the set of all vertices which can reach some vertex involved in some cycle in A)[1].

Definition Given a set of positive fair edges, n conditions of the form $F^\infty(S_i) + G^\infty(T_i)$, and a current set of states S, the ***forward bad-path operator*** returns a set of states $x \in S$, such that:
1. for each set of positive fair edges, there is a path in S, starting at x, and reaching some edge in the set, and
2. for each condition $F^\infty(S_i) + G^\infty(T_i)$, either $x \in T_i$, or there is a path in S, starting at x, and reaching a state in S_i.

The computation for this operator is $FP(S) = \left(\prod_{i=1}^{n} (R^*(x, S \cap S_i) + T_i)\right) \wedge \left(\prod_{j=1}^{p} R^*(x, E_j)\right)$, where n is the number of CFCs, S_i and T_i are the sets in the i-th CFC $F^\infty(S_i) + G^\infty(T_i)$, 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, except that the direction of the path is the reverse. The computation for the backward bad path operator is $BP(S) = \prod_{i=1}^{n} (R^*(S \cap S_i, y) + T_i) \wedge \prod_{j=1}^{p} R^*(E_j, y)$.

The algorithm for computing the set of all fair reachable states can now be described as follows.

1. Restrict the transition function $T(x, y)$ to the set of reachable states.
2. Remove negative fair edges from $T(x, y)$.
3. Let S_0 be the set of reachable states. Repeat until convergence,
  3.1 Apply forward stable set operator; $S_{i+1/2} = S^*(x, S_i)$.
  3.2 Apply forward bad-path operator; $S_{i+1} = FP(S_{i+1/2})$.
4. Let the set computed in step 3 be U. Compute the set of states which can reach U.

Theorem The above algorithm computes $Fair+$, the set of fair reachable states.

Proof Let the set returned by the above algorithm be W. To show $Fair+ \subseteq W$, it suffices to show $Fair \subseteq U$, where $Fair$ is the set of fair states. Let $x \in Fair$. Then, x is in some fair cycle. So, it is not deleted by forward bad-path and stable set operators. Since, x is a reachable state, we conclude $x \in U$. To show $W \subseteq Fair+$, it suffices to show $U \subseteq Fair+$. Now, let $x \in U$. We will show $x \in Fair+$. Assume to the contrary that x cannot reach a fair cycle. 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-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-th fairness constraint. We conclude that γ satisfies all positive edge constraints and all CFCs (γ satisfies the negative fair edges constraints since all the negative fair edges were removed). Hence, γ is fair. But this is a contradiction to x not being able to reach any fair cycles (QED).

There is another operator which can be used in step 3 of the above algorithm to quickly eliminate portions of the state space

1. The backward operators introduced in this section will be used in the next section.

in which no fair reachable states exist. Given a set of states S, the ***forward trim operator*** is defined by the following algorithm.

Perform the following until there are no more change:

1. For each $F^{\infty}(S_i) + G^{\infty}(T_i)$, if $\overline{S}_i \cap \overline{T}_i$ can only reach itself in the forward direction, let $S = S \cap T_i \cap S_i$.

2. For each set of positive fair edges $\{(a_1, b_1), \ldots, (a_k, b_k)\}$, if $\overline{\sum_{i=1}^{k} a_i} + \overline{\sum_{i=1}^{k} b_i}$ can only reach itself in the forward direction, let $S = S \cap \sum_{i=1}^{k} a_i \cap \sum_{i=1}^{k} b_i$.

The backward trim is defined by replacing all occurrences of the word "forward" in the above by the word "backward". One can easily show that the trim operators will not delete any fair reachable states.

5 Fair CTL

In this section, for completeness, we first define Computation Tree Logic (CTL) ([Cla86]). We then introduce fair CTL, and give a model checking algorithm for this logic.

5.1 CTL

Definition Let ***AP*** denote a set of atomic propositions, i.e. a set of symbols. The ***syntax of CTL*** is described as follows.

1) Any atomic proposition P is a CTL formula.

2) If f and g are CTL formulas, so are the following: $\bar{f}$, $f+g$, EXf, $E[fUg]$, and EGf.

To define the semantics of CTL, we need the notion of a Kripke structure.

Definition A ***Kripke structure*** on *AP* is a finite directed node-labeled graph, where the node labels are subsets of AP. We assume that every state in a Kripke structure has some successor. If a state *s* is labeled by $P \in AP$, we say that P is true at *s*. If a node is not labeled by some $P \in AP$, it is assumed that $\bar{P}$ is true there. A FSM *S* can be considered a Kripke structure over Σ by ignoring the edge labels. The atomic propositions are now the state names and the outputs produced at a state.

Definition The semantics of a CTL formula φ is defined in terms of a state *s* in a Kripke structure *S*. We say φ is true at *s* if one of the following holds:

1. $\varphi = P$ for atomic proposition P, and P is true at s.
2. $\varphi = \bar{f}$ for CTL formula *f*, not satisfied at *s*.
3. $\varphi = f+g$ for CTL formulas *f* and *g*, such that either *f* or *g* is satisfied at *s*.
4. $\varphi = EXf$ for CTL formula *f*, such that *f* is true at some next state of *s*.
5. $\varphi = E[fUg]$ for CTL formulas *f* and *g*, such that there exists a path π starting at *s*, where the first time *g* is true along π, *f* has been true for all previous times. Intuitively, this means that *f* is true until *g* becomes true.
6. $\varphi = EGf$ for CTL formula *f*, such that there exists a path π starting at *s*, where *f* is true everywhere along π.

We say φ holds for S if it holds at some initial state.

For ease of readability, we introduce the following syntactic abbreviations, with their intended meanings.

a. $AXf \equiv \overline{EX\bar{f}}$, meaning *f* is true at all successors of *s*.
b. $EFf \equiv E[trueUf]$, meaning there exists a path starting at *s* along which *f* finally becomes true.
c. $AFf \equiv \overline{EG\bar{f}}$ meaning for every path π starting at s, *f* is true at some state along π, i.e. it becomes true along π.
d. $AGf \equiv \overline{EF\bar{f}}$, meaning for every path π starting at *s*, *f* is true along all states of π, i.e. *f* is true at all states reachable from *s*.
e. $A[fUg] \equiv \overline{E[\bar{g}U\bar{f}\bar{g}]} \wedge \overline{EG\bar{g}}$ meaning that for all paths π starting at s, the first time g is true, f has been true for all previous times.

5.2 Fair CTL

As mentioned before, one is usually interested in restricting the set of paths considered in determining the truth of a formula φ in a Kripke structure *S*.

Definition Let $F^{\infty}f$ (***infinitely often*** *f*) hold for a path π iff the CTL formula *f* holds for an infinite number of times along π.

Let $G^{\infty}g$ (***almost everywhere*** g) hold for a path π iff the CTL formula $\bar{g}$ holds for only a finite number of times along π, i.e. after a finite amount of time g holds forever along π. These operators are sometimes called the ***infinitary operators***.

Remark Assume a Kripke structure S, a formula φ of the type $F^{\infty}f$ (or $G^{\infty}f$) where f is a CTL formula, and a path π in S, are given. One can calculate the set of states in which f is true, create a new atomic proposition Q, and assign Q to all nodes in which f is true. Now $F^{\infty}f$ (or $G^{\infty}f$) is true of π iff $F^{\infty}Q$ (or $G^{\infty}Q$) is true of π. Hence, we assume that only atomic propositions are used in conjunction with the infinitary operators.

Definition A ***general fairness constraint*** is any Boolean combination of F^{∞} and G^{∞}. A ***canonical fairness constraint (CFC)***[1] is a fairness constraint of the form $\sum_{i=1}^{m}\prod_{j=1}^{n}(F^{\infty}P_{ij}+G^{\infty}Q_{ij})$, where P_{ij} and Q_{ij} are atomic propositions.

Lemma ([Eme87]) Any general fairness constraint can be expressed as a CFC; but, the translation may be exponential.

However, [Eme87] argues that most practical fairness constraints can be represented efficiently using CFCs. In practice, usually a subset of the CFCs is supported, e.g. $\prod_{i=1}^{n}O^{\infty}P_i$, where O^{∞} is one of the two infinitary operators.

Definition A formula in the logic ***fair CTL*** is a pair (φ, Φ), where φ is a CTL formula and Φ is a canonical fairness constraint. The truth of (φ, Φ) is determined at a state s by only considering the paths which satisfy the fairness constraint Φ (these paths are called ***fair paths***).

5.3 Fair CTL Model Checking

[Eme87] introduced Fair CTL and gave an algorithm to model check fair CTL formulas. This algorithm relied on finding a ***fair strongly connected component*** of the graph, i.e. a SCC which completely contains a fair path. The first step in detection of fair SCCs finds all SCCs of the graph. Unfortunately, this operation is very time-consuming (on large graphs), even when BDDs are used ([Hoj92]).

[Bur92] first presented a BDD-based algorithm for a subset of fair CTL, where the fairness constraint is of the form $\prod_{i=1}^{n}(F^{\infty}P_i)$. In this paper, we present an algorithm for full fair CTL, which uses the fair reachable state computation of section 4 as a subroutine.

Remark To compute fair-states for a system S with fairness constraints of the form $\sum_{i=1}^{m}\prod_{j=1}^{n}(F^{\infty}P_{ij}+G^{\infty}Q_{ij})$, we compute fair-states for each constraint $\prod_{j=1}^{n}(F^{\infty}P_j+G^{\infty}Q_j)$, and then take their union. So, for the remainder of this section, we assume the fairness constraints are of the form $\prod_{j=1}^{n}(F^{\infty}P_{ij}+G^{\infty}Q_{ij})$.

To model check a fair CTL formula, we start from the leaves (which are atomic propositions), and compute the set of states satisfying each subformula. If some initial state is in the final set of states, the formula is satisfied. For example to compute the set of states satisfying $f+g$, we first compute the set of states satisfying f, the set of states satisfying g, and take their union. If an initial state is in this union, the formula is satisfied.

Notation Let $S(p)$ denote the set of states satisfied by a CTL formula p. Let $T(x, y)$ denote the transition graph of a system S.

In order to present a MC algorithm for fair CTL, we need to present algorithms for six cases: AP P, $p+q$, $\bar{p}$, EGp, EXp, and $E[pUq]$. In general, p and q are CTL formulas, and we assume the sets of states satisfying p and q are known. Let F be the set of fair reachable states of the system. The following algorithm performs the task.

1. We are abusing notation a little here. The remark in the next section will justify this abuse.

1. AP P. Return $S(P) \cap F$.
2. $p+q$. Return $S(p) \cup S(q)$.
3. $\bar{p}$. Return $\overline{S(p)}$.
4. EGp. Compute the set of fair-states, which also satisfy EGp, using following algorithm.
4a. Restrict $T(x, y)$ to $S(p)$. Call the result $T'(x, y)$.
4b. Run fair reachable state computation on this new graph $T'(x, y)$ with the same fairness constraints.
5. EXp. Take the intersection of the set of states whose next states satisfy p, with F.

$$U(x) = \exists y (T(x, y) \wedge S(p)(y) \wedge F(y))$$

6. $E[pUq]$. Compute the set of fair-states satisfying $E[pUq]$, using the following fixed-point computation.

$$U_0 = S(q) \cap F$$
$$U_i(x) = U_{i-1}(x) + \exists y (T(x, y) \wedge S(p)(x) \wedge U_{i-1}(y))$$

To prove the correctness of the above algorithm, we need one lemma.

Lemma Assume a fair path π is given. Then, every state along π is a fair reachable state.

Proof The set of infinitely occurring states of the path satisfy the fairness constraints. For any state s along π, consider the path π', obtained by restricting π to start at s. The set of infinitely occurring states of π and π' are the same. Hence, π' satisfies the fairness constraints, and therefore is a fair reachable state (QED).

Theorem The above algorithm is correct.

Proof We prove the correctness of the last three operations.

1. EGp. Let s satisfy EGp. Then, there exists a path π starting at s, along which p is always true. If we restrict the graph to states where p is true, π is still a fair path of this new graph. Hence, the algorithm returns all states s which satisfy EGp. To see that it does not return any other state, note that it returns fair reachable states of a graph, where p holds everywhere. Hence, all fair paths it considers satisfy EGp.

2. EXp. If s satisfies EXp, then there exists a fair path starting at s whose next state t satisfies p. By the above lemma, $t \in F$. Conversely, if s is such that for one of its next states t, $t \in F$ and $t \in S(p)$, then s satisfies EXp. So, states whose next states are in $S(p) \cap F$ are exactly the states satisfying EXp. This is the set of states in step 5.

3. $E[pUq]$. Let s satisfy $E[pUq]$. Then, there is a path from s to a fair reachable state t such that q holds at t, and p holds at the states between s and t along the path. The above computation will hence include s, since $t \in S(q) \cap F$, and each $U_i \cap \overline{U_0}$ produced satisfies p. Moreover, any state returned by the computation satisfies $E[pUq]$ (QED).

Remark All of the steps of the above algorithm can be efficiently implemented using BDDs.

6 Language Containment

In this section, we are concerned with the ω-automata specifications. 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, $L(S) \subseteq L(T)$. 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 [Var86]. [Var86] suggested specifying both the system and the property using Buchi automata, and went on to give an algorithm for language containment in this environment. The usual method to verify that $L(S) \subseteq L(T)$, where $L(S)$ and $L(T)$ are the languages of the system and property respectively, is by checking that $L(S) \cap \overline{L(T)}$ 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 ([Sis87], [Saf89]).

[Kur87b] introduced 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 ([Har90]). The advent of BDDs allows for handling very large state spaces. [Hoj92] and [Hoj93b] gave various efficient BDD-based algorithms for language containment and debugging in the L-environment.

In this section, we introduce a new language containment based formal verification environment, which contains the L-environment as a subset. The advantage of this environment is that specifications can be more compact, however, the algorithms remain as efficient as the algorithms for the L-environment. Specifically, if the specifications come from the L-environment, the algorithms reduce to those for the L-environment. As was mentioned before, the system is modeled using edge-Streett automata, whereas the properties are modeled using edge-Rabin automata. We call this environment the ***eReS-environment***.

Edge-Rabin automata have several useful properties. First, the set of deterministic edge-Rabin automata accept the whole set of ω-regular languages[1]. In general, if an automaton is deterministic, complementation is much easier. Since deterministic edge-Rabin automata can be compactly complemented into edge-Streett automata, we reduce our language containment check to checking that the language of an edge-Streett automaton, formed by taking the product of the complement of the property and the system, is empty. We first define edge-Rabin automata, and show how one can complement a deterministic edge-Rabin automaton into an edge-Streett automaton. We then present the language emptiness check for edge-Streett automata. We conclude by discussing some expressiveness results about our environment.

6.1 Edge-Rabin Automata

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

1. Positive fair edge constraint, which is an edge of A.
2. Negative fair edges constraint, which is a set of edges of A.
3. CFC, which is of the form $F^{\infty}(S) \wedge G^{\infty}(T)$, where S and T are sets of states.

An ω-string x of A is accepted ($x \in L(A)$) if there is a run r of x in A such that at least one of the fairness constraints is satisfied, i.e.

1. r follows some positive fair edge ∞-often, or
2. there is a set of negative fair edges E, such that r does not follow any of the edges in E ∞-often, or
3. for some CFC, say the *i*-th one, $inf(r) \cap S_i \neq \emptyset$ and $inf(r) \subseteq T_i$.

Theorem For every deterministic edge-Rabin automaton R, there is a deterministic edge-Streett automaton S with the same state space and no more fairness constraints, which accepts the complement of the language of R.

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 $F^{\infty}(\overline{T_i}) + G^{\infty}(\overline{S_i})$. Let $x \in L(S)$. Since S and R are both deterministic, x has a unique run in both S and R. Call it r. We will show that r is rejected in R. Assume not. Then, r satisfies one of the three acceptance conditions in the definition of edge-Rabin automata. However, x has the same run in S, i.e. r. It is easy to see that r must be rejected in S; contradiction (QED).

Remark When BDDs 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.

6.2 Language Emptiness Check for Edge-Streett Automata

The language of an edge-Streett automaton is not empty iff there are no fair paths. Hence, one way to check for language emptiness is to use the fair reachable states computation: this returns a non-empty set precisely when the language of the automaton is not empty. However, the fair reachable state computation returns a super-set of the ***fair states*** (denoted $Fair$), which are states involved in some fair cycle. In language containment, we are only interested in these.

Definition Let $Fair\pm$ denote the set of states which can reach some fair cycle and are reached from some fair cycle.

Note $Fair \subseteq Fair\pm \subseteq Fair+$, so $Fair\pm$ is a better approximation to $Fair$. The following algorithm computes a subset of

1. Deterministic L-automata cannot express all ω-regular languages, however, one can define any ω-regular language by taking an intersection of a finite (possibly large) number of languages expressed by deterministic L-automata ([Kur87b]).

$Fair\pm$, containing $Fair$ (the proof is similar to the proof of the correctness of the algorithm for fair reachable states computation).

Remove negative fair edges.
Start with reachable states.
Repeat until convergence:
1a. Apply forward bad-path.
1b. Apply forward stable set.
1c. Apply forward trim.
2a. Apply backward bad-path.
2b. Apply backward stable set.
2c. Apply backward trim.

In a later section, we will modify this algorithm by checking for easy failures.

6.3 Expressiveness of Edge-Streett Automata

We present two expressiveness results. First, we discuss why the eSeR-environment enjoys a sense of maximality. Second, we compare the compactness of specifications in the eSeR-environment to that in the L-environment ([Kur87b]).

The heart of the language containment check is an emptiness check for edge-Streett automata. One may ask if it is possible to make the environment even more powerful without sacrificing too much efficiency (the language emptiness check for edge-Streett automata can be done in small polynomial time). The next most natural extension is to allow 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. Since the sum of a set of $F^{\infty}(S_i)$'s is just equivalent to $F^{\infty}(\bigcup_i S_i)$, only one F^{∞} suffices. It remains to answer whether allowing more G^{∞}'s will sacrifice efficiency. [Eme87] basically answered this question (they were looking at a slightly different problem).

Theorem The problem of determining whether the language of an automaton with acceptance conditions of the form $\prod_i (G^{\infty}(S_i) \wedge G^{\infty}(T_i))$ is empty, is NP-complete (in the number of states and fairness constraints).

Proof (sketch) 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 [Eme87], which was employed to show the fair state problem with fairness constraints of the above type is NP-complete. [Eme87] reduces 3SAT to a graph with fairness constraints of the above type. Now, the language of the automaton is non-empty iff the original formula is satisfiable (QED).

Since the language emptiness check for Streett automata is done in polynomial-time, the next natural extension to our environment makes the language emptiness check, which is the most important operation in formal verification, NP-complete. This gives the eSeR-environment a sense of maximality.

Remark [Hoj93c] discusses several compactness issues of the eSeR-environment. The most important result is that translating the specification from eSeR-environment to the L-environment ([Kur87] can be exponential in the worst case.

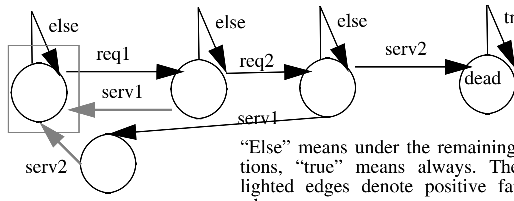
7 Complementary Nature of MC and LC

We give a classification of properties, which constitutes a different way of looking at property specification methods, such as CTL and edge-Rabin automata. We argue that, using this view, LC and MC are somewhat complementary.

7.1 A Property Classification

The properties one might want to verify about a system can be divided into three sets: string properties, language properties, and system properties.

1. ***String Properties***. These properties must be satisfied by every trace of the system, i.e every ω-string in the language of the system. An example is: for every trace of the system, if request 1 is pending when request 2 is generated, then request 1 is serviced before request 2. The edge-Rabin automaton recognizing this property is given below. In the diagrams below, a highlighted box around a set of states S denotes a constraint of the form $F^{\infty}(\bar{S}) \wedge G^{\infty}(True)$.



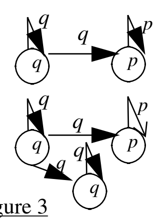
"Else" means under the remaining conditions, "true" means always. The highlighted edges denote positive fair edge edges.

Figure 2

Much is known about the hierarchy of specification languages for traces. At the highest level are non-deterministic ω-automata, such as non-deterministic Buchi automata ([Cho74]). The set of languages defined by these automata is known as the ***ω-regular languages***. ***Linear Temporal Logic (LTL)*** is another method to specify string properties. LTL is properly contained in ω-regular languages. [Eme90] gives a survey of such expressiveness results.

2. ***Language Properties***. These are properties of the languages, not expressible as string properties. For example, one might wish to verify that if strings x and y are in the language then string z is also in the language. An example is the CTL property $P = AGEFp$, which says for all reachable states on fair paths, we can reach a state where p is true. Any system having the above property P has the following language property Q: Q is the set of languages L such that if $yz \in L$, where y is a finite string and z is an infinite one, then the string $yw \in L$, where p occurs in w.

3. ***System Properties***. These properties distinguish between systems, possibly having the same language. An example is the above CTL property $P = AGEFp$, which distinguishes between the systems shown in figure 2, having the same language.



The CTL property $AGEFp$ holds for the first system, but it does not hold for the second. Note that both systems have the same language. Hence, CTL has the power to distinguish between different implementations of the same behavior.

Figure 3

Some properties may be a combination of properties of the above categories. For example, if strings x and y having string property P belong to language L, then string z belongs to L. We won't be concerned with such properties mainly because they have not been encountered in practice.

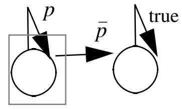
7.2 MC versus LC

To compare MC and LC, we first need to define expressiveness of a specification language.

Definition A ***specification language (S, A)*** is a set S (usually infinite) of finite strings over some alphabet, and an algorithm A to decide whether a system satisfies a member of S. An example is the set of all CTL formulas as S, and a CTL model checking algorithm as A. In this case, the alphabet is the set of all outputs of all systems, plus symbols used in CTL formulas. We assume that systems are given in the CS model. Other common finite state models can be used instead.

Definition Given two specification languages (S_1, A_1) and (S_2, A_2), we say S_2 is at least as ***expressive*** as S_1, if for any $x \in S_1$ there exists some $y \in S_2$ such that $A_1(x) = A_2(y)$, for any system S. The intuition is that given a property p_1 in S_1, we can come up with a corresponding property p_2 in S_2, such that for any system S, p_1 holds of S iff p_2 holds of S.

In both MC and LC, an important subclass of trace properties, called invariance properties, is expressible. ***Invariance properties*** are of the type "no bad thing happens in any state of the system." Invariance properties are expressed by formulas of type AGp in CTL (p may be a Boolean formula of atomic propositions) or as edge-Rabin automata of the type shown in figure 4.



This edge-Rabin automaton expresses the invariance property that in all states p is true. If we ever visit the second state, we will stay there forever, and the resulting string is rejected.

Figure 4

Moreover, some simple ***liveness properties***, i.e. properties of the form "some good thing will happen in the future", are also expressible by both methods.

However, there are string properties expressible by ω-automata not expressible by CTL formulas. An example ([Eme90]) is the property "at all even times p holds." There are other properties, usually involving sequencing of events, which are very hard or impossible to express in CTL. An example is "if $req1$ happened before $req2$, then $serv1$ should happen before $serv2$," for which the edge-Rabin automaton is shown in figure 5. Properties depending on sequencing of events occur often in practice. Examples are memory systems and pipelined computers. It is our experience that even when a property involving sequences of events is expressible in both CTL and edge-Rabin automata, it is harder to find the correct CTL formula for it.

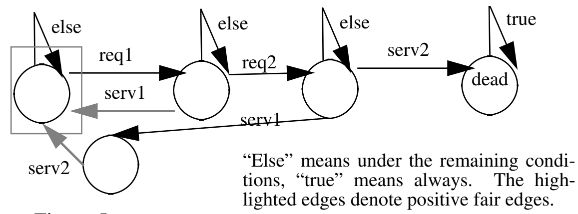


Figure 5

On the other hand, no language or system properties are expressible by ω-automata. In general, properties involving the existential quantifier of CTL are not expressible by ω-automata. These properties are significant, because sometimes we want to make sure that our system will be able to reach a safe state from any state of the system. The CTL formula $AGEFp$, which involves existential quantification, expresses this property.

We believe LC and MC provide different, useful capabilities to the user, and therefore, are somewhat complementary. By combining the two environments, we gain more expressive power and ease of expression; a larger set of properties can be more easily specified and verified. It remains to determine how much of an overlap there is in the useful expressive powers of CTL and ω-regular languages.

Remark It may also be the case that debugging with CTL properties is easier, since the debugger unfolds the formula one operator at a time ([Hoj93b]). Language containment, however, gives all the debugging information back to the user in one bunch. Hence, at every point in debugging CTL formulas, less information is fed back to the user.

8 Early Failure Detection

We present techniques for early failure detection (***EFD***). We first present a classification of errors for LC, and show how each can be computed. In most cases where a property fails, we do not need to compute the full set of reachable states to find the error. Indeed, since the notion of an easy error in many cases corresponds to an error which is reproducible in a few steps, we expect to compute only a few reachability steps before the error is found. We combine this idea with the error classification to given an algorithm for EFD during LC. The idea of avoiding the full reachability set is used then to give a EFD algorithm for ACTL.

8.1 Early Failure Detection for LC

In LC, a failure is an ω-string accepted by an edge-Streett automaton. Given the transition function with negative fair edges removed, and a subset R of the reachable states, we define the following set of fair cycles.

1. ***Cycles of the first kind***. These are cycles in R 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 in R intersecting $\bigcap_i S_i$, and satisfying the positive edge constraints.

3. ***Cycles of third kind***. Any fair cycle in R 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 R, with the fairness constraints being as follows:

1. Positive edge constraints.

2. A constraint of the form $F^{\infty}(\bigcap_i S_i)$.

If a non-empty set is returned, we have some fair behavior.

Theorem If there are no cycles of the first or second kind, the main computation can be restricted to $\overline{\bigcap_i T_i} \cap \overline{\bigcap_i S_i} \cap R$.

Proof We can delete the states in $\bigcap_i S_i$ and $\bigcap_i T_i$ from consideration, because there are no cycles of the first or second kind, and hence there cannot be any fair cycles involving states in $\bigcap_i S_i$ or $\bigcap_i T_i$.

To avoid computing the full reachability set, as we compute the set of reachable states, we can perform a cheap test which may find some errors. The following algorithm is an implementation of this idea. We start with the current set equal to the set I of initial states.

Enhanced Early Failure Detection

1. Using breadth-first search (BFS), compute the set of states R, reachable i n D or fewer steps from the current set.
2. Check whether there are any cycles of the first kind in R. If any cycles are found, stop, and call the debugger.
3. If none is found, set the current set equal to R, go to step 1, continuing the BFS for another D steps or until all reachable states are computed.

Intuitively, the algorithm takes a few BFS reachability steps (D steps), and then checks whether there are any (easily found) bad cycles of the first kind in this set. Since easy errors are almost always of the first kind, and since checking for these is less computationally expensive than for checking cycles of the second kind, we only invoke EFD here with cycles of the first kind (although obviously other fair cycles could be checked as well). The process continues until an error is found, or the full set of reachable states is computed.

8.2 Early Failure Detection in Fair CTL

So far, we do not know how to extend EFD techniques to general CTL formulas. However, the technique described in subsection 8.1 can be applied to ACTL formulas, where the idea is to take a few reachability steps, and then check whether the formula is satisfied. The trouble with formulas involving existential quantification is that the reachable states not yet visited may be “witnesses” for the existential quantification. This problem does not show up with ACTL formulas.

Lemma Let S be a subset of reachable states R. Let ϕ be an ACTL formula, interpreted over CFC Φ. Let $s \in S$. Then, if ϕ does not hold at s (over Φ) in S, then ϕ does not hold at s (over Φ) in R.

Proof We use induction over the definition of CTL formulas. Assume ϕ does not hold at s in S. Note that ϕ is interpreted over fair paths, and the fair paths in S are a subset of the fair paths in R.

1. $\phi = P$, P some proposition. If there were no fair paths starting at s, then any formula would hold at s. Hence, there must exists some fair path in S starting at s. Since P is not a label of s, ϕ does not hold in S, and hence not in R either.

2. $\phi = f + g$, where f and g are CTL formulas. By the same reasoning as in case 1, neither f nor g hold at s in S. By the inductive assumption, neither f nor g hold of s in R. The conclusion follows.

3. $\phi = f \wedge g$. Similar to case 2.

4. $\phi = AGf$. Hence, there is some state x reachable from s along a fair path π, at which f does not hold. By inductive assumption, f does not hold of x in R. Since π is a fair path in R, ϕ does not hold of s in R.

5. AXf. Proof is similar to case 4.

6. AFf. Proof is similar to case 4 (QED).

Since we do not have any way of quickly determining whether an ACTL formula is not satisfied, the check should be applied more sparingly than for edge-Rabin automaton. The problem of extending quick failure checks to ACTL formulas remains open.

Remark One important topic not dealt with in this paper is debugging. When early failure detection is used, not all fair states are found. Hence, a debugger must be able to deal with partial information. [Hoj93b] discusses techniques for finding errors using partial information, as well as techniques for finding short error traces.

9 Hierarchical Verification

The design process can be done as a hierarchical activity. The designer first abstracts the design, and proves some properties about it. Next, the designer adds more detail. For example, a module is implemented by three modules. By definition, a subsystem implements another if the behavior of the implementation is contained in the behavior of the specification at a higher level. This means that the 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. [Kur92] presents an algorithm for language containment of two L-processes (this comes up in the L-environment because of use 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, where S_2 is deterministic. Using the following two lemmas, the problem of containment of $L(S_1)$ by $L(S_2)$ is reduced to a set of language containments between edge-Streett and edge-Rabin automata.

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 $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$ or $F^{\infty}(S_i) + G^{\infty}(T_i)$.

Lemma $L(S_1) \subseteq L(S_2)$ iff $L(S_1) \subseteq L(S_i)$, for $1 \le i \le n$, 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 (=>) Assume $L(S_1) \subseteq L(S_2)$. 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, $L(S_1) \not\subset L(S_2)$, which is a contradiction.

(<=) Assume to the contrary that $L(S_1) \not\subset L(S_2)$. Let x be a string in $L(S_1)$ and not in $L(S_2)$. 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 (QED).

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

Proof We have to show the transformation for each kind of fairness constraint. We have three cases:
1. The constraint is a negative fair edge. Create an edge-Rabin automaton with the edge being a negative fair edge constraint.
2. The constraint is a set of positive fair edges. Create an edge-Rabin automaton with these edges being a positive fair edges constraint.
3. The constraint is of the form $F^{\infty}(S_i) + G^{\infty}(T_i)$. Create an edge-Rabin automaton with two CFCs: $F^{\infty}(Q) \wedge G^{\infty}(T_i)$ and $F^{\infty}(S_i) \wedge G^{\infty}(Q)$, 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 (QED).

Remark Note that the algorithm for language containment between two edge-Streett automata is rather efficient. For checking whether $L(S_1)$ is contained in $L(S_2)$, 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 n 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 CFCs, [Saf92]'s algorithm returns a deterministic Rabin automaton with $n2^{h \log h}$ states and $h+1$ CFCs accepting the same language. Although this algorithm is exponential in h, if h is 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 (exponential) algorithm for determinizing a non-deterministic Streett automaton, which returns a deterministic Rabin automata. One can translate an edge-Streett automaton into a Streett automaton, using techniques similar to the node-recur transform of [Kur87a]. Hence, the problem of testing for language inclusion of an edge-Streett automaton into another edge-Streett automaton can be reduced to testing the language inclusion of an edge-Streett automaton into a deterministic Rabin automaton, for which we have presented algorithms before.

Remark In practice, many specifications are effectively deterministic, i.e. have deterministic transition structures, since output non-determinism can be used for the necessary abstractions in many cases.

10 Conclusion

An environment for formal verification using language containment and fair CTL model checking was presented. We have

1) presented a powerful concurrency modeling scheme called the combinational/sequential concurrency model,

2) presented ways for specifying fairness constraints,

3) given algorithms for fair CTL model checking and language containment in the eSeR-environment using a subroutine for computing fair reachable states,

4) argued why having both automata and CTL in the same environment is useful, and

5) given techniques for finding easy failures quickly without doing the full computation.

All of our algorithms are BDD-based, and hence are expected to handle large graphs. The full set of algorithms described in this paper have been or are being implemented in the Berkeley's verification and synthesis program, HSIS.

Acknowledgment

This paper is a composite and extension of the material in references [Hoj92], [Hoj93a], and [Hoj93c]. We gratefully acknowledge the many contributions of our co-authors, R. Kurshan, T. Shiple, V. Singhal, and H. Touati. In addition, the generous guidance of R. Kurshan over the last three years is much appreciated. This work has been supported by the SRC under grant 93-DC-008, NSF/DARPA Grant MIP-8719546, Intel, Motorola, AT&T, DEC, and the California Micro Program.

References

[Bry86] R. E. Bryant, "*Graph Based Algorithms for Boolean Function Manipulation*", IEEE Trans. on Computers, C-35(8):677-691, August 1986.

[Bur92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. Dill, L. J. Hwang "*Symbolic Model Checking: 10^{20} states and beyond*", Information and Computation, June 1992.

[Cho74] Y. Choueka, "*Theories of Automata on ω-Tapes: A Simplified Approach*", Journal of Computer and System Sciences 8, 117-141, 1974.

[Cla86] 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), pp.244-263, 1986.

[Eme87] E. A. Emerson, C. L. Lei, "*Modalities for Model Checking: Branching Time Logic Strikes Back*", Science of Computer Programming 8, 275-306, Elsevier Science Publishers, 1987.

[Har90] Z. Har'El and R. P. Kurshan, "Software *for Analytical Development of Communication Protocols"*, AT&T Journal, Jan 1990, 45-59.

[Eme90] E. A. Emerson, "*Temporal and Modal Logic*", Handbook of Theoretical Computer Science, editor J. van Leeuwen, Elsevier Science Publishers B.V., pp. 995-1072, 1990.

[Hoj92] Ramin Hojati, Herve Touati, Robert P. Kurshan, Robert K. Brayton, "*Efficient ω-Regular Language Containment",* Computer-Aided Verification, 1992.

[Hoj93a] R. Hojati, Thomas P. Shiple, R. K. Brayton, R. P. Kurshan, "A *Unified Approach to Language Containment and Fair CTL Model Checking"*, Proceedings of Design Automation Conference, June 1993.

[Hoj93b] R. Hojati, R. K. Brayton, R. P. Kurshan, "BDD-Based *Debugging of Design Using Language Containment and Fair CTL"*, Proceedings of the Conference on Computer-Aided Verification, Elounda, Crete, Greece, To appear, June 1993.

[Hoj93c] R. Hojati, Vigyan Singhal, R. K. Brayton, "Edge-Streett/Edge-Rabin *Automata Environment for Formal Verification Using Language Containment"*, Submitted to the International Conference on Computer-AIded Design, 1993.

[Kam91] T. Kam, R. Brayton. "*Multi-valued Decision Diagrams*", Electronics Research Laboratory, University of California, Berkeley, Memorandum No. UCB/ERL, M90/125, 1990.

[Kur87a] R. P. Kurshan, "Complementing *Deterministic Buchi Automata in Polynomial Time"*, Journal of Computer and System Sciences, volume 35, 1987, 59-71.

[Kur87b] R. P. Kurshan, "*Reducibility in Analysis of Coordination*", In LNCIS, volume 103, pages 19-39, Springer-Verlag, 1987.

[Kur92] R. P. Kurshan, "Automata-Theoretic *Verification of Coordinating Processes"*, UC Berkeley notes, 1992.

[Rab72] M. O. Rabin, "Automata *on Infinite Objects and Church's Problem"*, Regional Conference Series in Mathematics, volume 13, 1972, American Mathematical Society, Providence, Rhode Island, 19-39.

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

[Saf92] S. Safra, "Exponential *Determinization for ω-Automata with Strong-Fairness Acceptance Condition"*, STOC, 1992.

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

[Tou90] H. J.Touati, H. Savoj, B. Lin, R. K. Brayton, A. S. Vincentelli, "*Implicit State Enumeration of Finite State Machines Using BDDs*", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 130-133, Nov. 1990.

[Tou91] H. Touati, R. Kurshan, R. Brayton. "*Testing Language Containment of* ω*-Automata Using BDDs*", International Workshop on Formal Methods in VLSI Design, 1991.

[Var86] M. Y. Vardi and P. L. Wolper, *An Automata-Theoretic Approach to Program Verification*, Logic in Computer Science, 332-334, 1986.