# BDD-Based Debugging Of Designs Using Language Containment and Fair CTL

Ramin Hojati (UC Berkeley)
Robert K. Brayton (UC Berkeley)
Robert P. Kurshan (AT&T)

## Abstract[1]

Formal design verification should be used to reveal bugs early in the design cycle. A tool exhibiting counter-examples (a debugger) is therefore essential. We describe debugging techniques for two important approaches to formal design verification: model checking using Computation Tree Logic ([Cla86]) and language containment using L-automata ([Kur90]).

The contributions of this work are:

1. A debugger for language containment (***LC***), using L-automata, which guarantees some minimality criteria.
2. A debugger for fair CTL ([Eme87]) model checking based on the algorithms for LC debugging.
3. Enhanced early failure detection (***EFD***) ([Hoj92]) algorithms for LC, and simple EFD algorithms for a subset of CTL formulas, called ACTL (for All CTL) in positive normal form.

All algorithms are based on Binary Decision Diagrams (BDD's). The algorithms have been partly implemented. We comment on some experimental results.

## 1 Introduction

Formal design verification is the process of proving "what one has specified is what was wanted," by checking properties of the design. One of the most important uses of formal verification may well be as an "intelligent" simulator for debugging the design specification. In this use, the designer begins writing the specification and at a certain point enters the debugging phase (at the same point where he might want to simulate). It is expected that the design at this early stage is not bug free. The formal verifier must then present examples of bad behavior. But with formal verification, since the verifier has the ability to determine all bad behaviors, it can try harder to present the user with short counter-examples, which can hopefully be used more easily by the designer to locate the error.

However, if this capability is to replace simulation, we need also to report easily detectable errors without performing a lot of computation. Thus in designing a debugger for formal verification, we are motivated by two goals:

1. to report short traces of examples of bad behavior, and
2. to report easily detectable errors quickly.

Model checking (***MC***) using CTL and language containment using L-automata are two methods for property specification for which efficient verification algorithms exist. [Hoj93] described a technique for integrating fair CTL (an expressive member of the CTL family of logics) and LC using L-automata in one environment. In this paper, we first describe debugging algorithms for LC (section 2, 3 and 4), thereby building a debugging environment. We then use this environment and additional ideas to describe a debugger for fair CTL.

An error trace in LC consists of a path to a "bad cycle" and the "bad cycle" itself. We present BDD-based algorithms which report an error trace such that the path to a bad cycle is minimum. They also heuristically minimize the length of the bad cycle reported. We prove here, the problem of finding a shortest bad cycle is NP-complete.

As for CTL (from hereon, by CTL we mean fair CTL), we unwind the CTL formula one operator at a time, giving debugging information about the topmost operator. This has the advantage of not giving the user an overwhelming amount of information at one time. We hope that this will prove useful as more experiments are done with debugging techniques. When existential quantification is used, the user has to interact with the debugger to find the source of error. For ACTL formulas the debugger is capable of returning the debug trace automatically, although the option to interact is provided.

An important feature of the algorithms presented in [Hoj92] for LC using L-automata is early failure detection (EFD). Here, we enhance these techniques by avoiding the full reachability set for simple errors. We apply the same ideas to ACTL formulas, described in ***positive normal form*** (***PNF***). Efficient algorithms for EFD are very important, since in practice the tool will be run many more times with properties which fail than with those which pass.

A debug trace can be used in another context: timing verification. In some situations, a property holds if realistic timing constraints are considered. An example of a timing constraint is "the system can stay in state $s$ for only 2 ns." A method used in [Bal92] and [Alu92] is not to consider all timing constraints. If the property holds of the system when only a subset of timing constraints is considered, we are done. Otherwise, the debugger is called to find a counter-example. If the error trace is a counter-example even when all timing constraints are considered, we are again done. Otherwise, enough additional timing constraints are introduced so that the error trace is no longer a trace of the system. The check continues with this new subset of timing constraints.

In the rest of this section, we explain how formal verification can be cast as language containment, set up the language containment problem using L-processes and L-automata, and describe the rest of the paper.

### 1.1 Design Verification Using Language Containment

In this section, we use an example to show how one can cast property checking as language containment. Consider the following Moore machine, which simply ouputs the input it received during the last clock cycle.

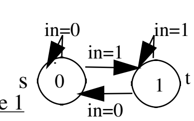


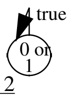

Figure 1

This Moore machine ouputs its input after exactly one clock cycle. The initial state is $s$.

The outputs of this FSM are 0 and 1. In design verification, one usually models the environment as well as the sys-

1. The first author is supported by the Semiconductor Research Corporation under grant 91-DC-008.

tem. In this case, as the environment, one can use the following automata which can produce a 0 or 1 non-deterministically at any time.




This automaton which serves as the environment for FSM in figure 1, can produce a 0 or 1 at any time non-deterministically (randomly).

Figure 2

We call the above model, which includes a model for the environment, an ***(abstract) model***. In general, we may have to abstract away some details of the system in order to make the verification task more tractable. The alphabet of the model, $\Sigma$, is the set $\{(0,0), (0,1), (1,0), (1,1)\}$, which is obtained by taking the Cartesian product of the outputs of the system ($\{0,1\}$) and the ouputs of the environment ($\{0,1\}$). In general, the alphabet of a model is the Cartesian product of outputs of the environment with the observable outputs of the system.

Since the above two automata are being executed simultaneously, we need to associate a concurrency model to them. Our concurrency model is the ***selection/resolution*** model ([Agg83]). According to this model, at each point in time, the system is at some global state, which consists of each process in some local state. Each process chooses an output possible from its state non-deterministically. This step is called ***selection***. During ***resolution***, each process chooses a next state based on the global selection, i.e. outputs of each individual machine during selection. This process continues indefinitely. Hence, each possible trace of the model is an infinite ($\omega$) string of symbols over the alphabet of the model. An example of a possible trace of the above system is $(0,0), (0,1), (1,1), (1,0), (0,1), \ldots$. We define the ***language of the model*** to be the set of all possible (infinite) traces it can produce according to the selection/resolution model.

Assume, we want to verify that the FSM produces a 1 in the next clock after the environment has produced a 1. The language $L(A)$, defined over $\Sigma$, describes this property:

$$L(A) = \{x = (x_0, x_1, x_2, \ldots) \mid x_i = (\text{-}, 1) \rightarrow x_{i+1} = (1, \text{-})\}$$

where "-" denotes any symbol. To show the system has the above property, it suffices to check that each string in the language of the system is of the above type, i.e. $L(S) \subseteq L(A)$, where $L(S)$ denotes the language of the system. Hence, property verification is reduced to verifying that the language of a system is contained in the language of a property. In the next section, we extend this modeling scheme and present a method for specifying $L(A)$.

### 1.2 L-process/L-automata Environment

Sometimes, abstraction introduces unwanted behavior. An example is in modeling delays. Assume we want to specify that the system can stay at some state $s$ for some finite but unknown amount of time. This is done by having a self-loop at state $s$. However, we want to exclude the behavior that the system can stay at $s$ forever. To do so, we introduce L-processes.

*Definition* An ***L-process*** ([Kur90]) is a non-deterministic transition non-deterministic output Moore machine, i.e.
1. it can produce a set of outputs non-deterministically from any of its states.
2. a set of transitions may be taken during resolution, one of which is chosen non-deterministically.
There are also two extra fields: a set of edges known as ***recur edges***, and a set of subsets of states known as ***cycle sets***. Formally, an L-process is defined by a tuple $(Q, \Sigma, T, I, O, R, C)$, where $Q$ is the set of states, $\Sigma$ the alphabet, $T$ the transition relation, $I$ the set of initial states, $O$ the output relation, $R$ the set of recur edges, and $C$ the set of cycle sets.

*Definition* Let $x = (x_0, x_1, x_2, \ldots)$ be an infinite string. A ***run r of x*** in an L-process $P$ is a set of states $r = (s_0, s_1, s_2, \ldots)$ such that $s_0$ is an initial state, and for all $i$, there is an edge between $s_i$ and $s_{i+1}$ labeled with $x_i$. Intuitively, a run corresponds to a path in an L-process, whereas a string corresponds to the labels of the path. Note that since an L-process can have non-deterministic transitions, there may be several runs for the same string.

*Definition* Let $r$ be a run of (infinite) string $x$. Then, ***inf(r)*** is the set of states occurring infinitely often in $r$. Since an L-process has a finite number of states, and since a run is infinite, $inf(r) \neq \emptyset$.

*Definition* Given an L-process $P$, a string $x$ is ***accepted*** if there exists a run $r$ of $x$ such that no recur edge is traversed infinitely often and $inf(r)$ is not contained in any of the cycle sets. These conditions, recur edges and cycle sets, imposed on the string are called the ***excepting conditions***. Intuitively, a string is accepted if it is a trace of the system and none of the excepting conditions is satisfied. The set of all strings accepted by $P$ is known as the ***language*** of $P$, and is denoted by $L(P)$.

*Remark* Note that one can use cycle sets and recur edges to except those behaviors where a process stays at some state forever. This provides a solution to the problem of modeling finite but unbounded delays.

*Definition* Let $P_1, P_2, \ldots, P_n$ be a set of L-processes. The language ***associated with*** $P_1, P_2, \ldots, P_n$ ***according to the selection/resolution model*** is the set of strings generated by $P_1, P_2, \ldots, P_n$ according to selection/resolution model, such that none of the excepting conditions of the individual machines are satisfied. Intuitively, we ignore recur edges and cycle sets, and run the processes in parallel according to the selection/resolution model. We then only accept those strings which have an accepting run in each individual machine. Note that a global run can be projected onto individual machines, since every global state is a tuple of individual states.

*Lemma* ([Kur90]) Let $P_1, P_2, \ldots, P_n$ be a set of L-processes defined over some alphabet $\Sigma$. Let $L$ be their language according to the selection/resolution model. Then, $L = \bigcap_{i=1}^{n} L(P_i)$.

*Definition* Let $P_1, P_2, \ldots, P_n$ be a set of L-processes, where each L-process is of the form $P_i = (Q_i, \Sigma_i, T_i, I_i, O_i, R_i, C_i)$. Define $P = (Q, \Sigma, T, I, O, R, C)$ to be the ***product*** of $P_1, P_2, \ldots, P_n$ as follows: $Q$ and $\Sigma$ are the cartesian products of $Q_i$'s and $\Sigma_i$'s respectively, $T = \prod_{i=1}^{n} T_i$, $I = \prod_{i=1}^{n} I_i$, $O = \prod_{i=1}^{n} O_i$, where the products are Boolean ANDs,

$R = \bigcup_{k=1}^{n} R_k$, and $C = \bigcup_{k=1}^{n} C_k$. We assume that sets and relations are represented by Boolean equations using techniques, such as the ones described in [Tou90]. In this case, cycle sets and recur edges of each individual machine become recur edges and cycle sets of the product machine.

*Theorem* 1 ([Kur90]) Let $P_1, P_2, \ldots, P_n$ be a set of L-processes defined over some alphabet $\Sigma$. The, the language of $P_1, P_2, \ldots, P_n$ according to the selection/resolution model is $L(P)$, where $P$ is the product of $P_1, P_2, \ldots, P_n$. So, $L(P) = \bigcap_{i=1}^{n} L(P_i)$.

Using the above theorem, we can specify a system by a single L-process. To specify properties, we use deterministic L-automata.

*Definition* ***Deterministic L-automata*** are syntactically the same as L-processes with two exceptions: first, the transition relation is deterministic, and second, there are no outputs. Semantically, the acceptance conditions of L-automata are complementary to those of L-processes, i.e. a string $x$ is accepted by an L-automaton if there is a run $r$ of $x$ such that some recur edge is traversed infinitely often or $inf(r) \subseteq C_i$, for some cycle set $C_i$. The language of an L-automaton is the set of strings accepted by it.

Hence, our paradigm for verification is reduced to checking that the language of an L-process is contained in the language of an L-automaton. As an example, consider the system of the last section where the L-process is obtained by taking the product of the FSM and its environment (it will be a two-state machine). The following figure gives the L-automaton corresponding to the property "the FSM produces a 1 in the next clock after the environment has produced a 1."

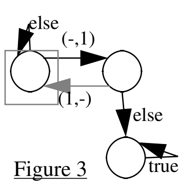



Figure 3

This L-automaton describes the property that if the environment generates a 1 in the current clock cycle, then the FSM generates a 1 in the next clock cycle. The highlighted edge is a recur edge. The box represents a cycle set. "true" means always, and "else" means for the rest of the symbols. Note only those strings are accepted where there is some run which stays in some cycle set forever, or goes through some recur edge infinitely many times.

*Definition* Let $A$ be a deterministic L-automaton. Define $P_A$ to be the L-process resulting from considering $A$ to be an L-process. We assume every output is possible from every state of $P_A$.

*Lemma* Let $A$ be a deterministic L-automaton. Then, $\overline{L(A)} = L(P_A)$ (the over-bar denotes complementation).

To test $L(S) \subseteq L(A)$, it suffices to check $L(S) \cap \overline{L(A)} = \emptyset$. By the above lemma, it suffices to check $L(S) \cap L(P_A) = \emptyset$. By theorem 1, $L(P) = L(S) \cap L(P_A)$, where $P$ is the product of $S$ and $P_A$. Hence, we need to check $L(P) = \emptyset$.

In summary, verification by property checking is reduced to the problem of checking whether the language of some L-process is empty ([Hoj92] offered several efficient BDD-based algorithms for this problem). Hence, an error trace is an infinite path which violates the excepting conditions of the L-process. An infinite path can be thought of as a cycle and a path from some initial state to that cycle, where the infinite behavior is obtained by traversing the cycle infinitely often. The cycle is such that traversing the cycle infinitely often violates the excepting conditions. Such a cycle is called a ***bad cycle***.

In this paper, we present a BDD-based algorithm which guarantees the path to a bad cycle is minimum, i.e. for any other error trace the path to its bad cycle is not shorter. We prove the problem of finding a shortest bad cycle is NP-complete, and therefore offer a heuristic solution.

Another feature of our debugger is that it requires little pre-processing: the only required input besides the transition relation is a set of states which contains all states involved in some bad cycle (these are called ***bad states***). As we will show, one can even relax this condition, and take as input a set of states containing a subset of bad states. The debugger still works; however, we are no longer guaranteed that the path to the bad cycle is minimum. This feature is important for EFD, where a subset of bad states is passed to the debugger.

### 1.3 Outline

In section 2, we introduce some preliminary concepts, and present some utilities. In section 3, we present debugging algorithms for LC. In section 4, we enhance EFD algorithms of [Hoj92] for LC, and describe how one finds error traces when EFD is used. Section 5 describes techniques for debugging of fair CTL formulas, using the utilities in section 3. We comment on some experimental results in section 6, and conclude in section 7.

## 2 Preliminaries

### 2.1 Basic Concepts and Definitions

*Definition* A ***cyclic strongly connected component (CSCC)*** of a graph $G$ is a SCC which contains some cycle. Note that some SCC's may contain only one state with no cycles. Such a SCC is called an ***acyclic SCC (ASCC)***.

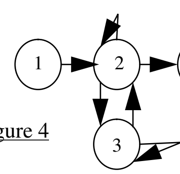



Figure 4

This graph contains three SCC's: {1}, {2,3}, {4}. The first one is acyclic, the last two cyclic.

The following theorem is the basis of all our LC algorithms.

*Theorem 2* Let $P$ be an L-process. Let $Q$ be $P$ with recur edges removed. Then, $L(P) = \emptyset$ iff for each CSCC $S_i$ of $Q$ there exists some cycle set of $P$ containing $S_i$.

In what follows, we work with the above graph $Q$ (represented by a BDD), with vertices $V$, and transition relation $T(x, y)$. Note that $x$ and $y$ denote the set of present and next state variables respectively. We let $\gamma_j$ for $j = 1, \ldots, n$ be the cycle sets of $Q$, and $I$ the set of its initial states.

*Definition* Let a graph $G$ be given. Let $A$ be a subset of vertices of $G$. We denote by $R^*(x, A)$ the set of vertices $x$ which can reach $A$ (in 0 or more steps), $R^*(A, y)$ the set of vertices $y$ reachable from $A$, $R_1(x, A)$ the set of vertices $x$

which can reach $A$ in exactly one step, $R_1(A, y)$ the set of vertices $y$ reachable from $A$ in exactly one step. Note that $R^*(x, A)$ and $R^*(A, y)$ are defined over variables $x$ and $y$ respectively. Note that the set $A$ can in particular be a single state $z$. All of the above sets are computed easily using BDD's.

### 2.2 Utilities

In this section, we present BDD-based algorithms for finding the SCC of a vertex, and the shortest path between two sets of states.

#### *2.2.1 Finding The SCC of A State*

The SCC of a vertex $z$ is computed by the formula: $SCC_z(y) = R^*(z, y) \wedge R^*(y, z)$, since every state returned by this computation is reached by $z$, and can reach $z$. Therefore, by definition it is in $SCC_z$. Conversely, every state of $SCC_z$ is in both $R^*(z, y)$ and $R^*(y, z)$. Hence, it is returned by the above computation.

*Lemma* Let $z$ be a state. The computation $C_z(y) = R^*(R_1(z, y), y) \wedge R^*(y, z)$ returns the empty set if $z$ is an ASCC. Otherwise, it returns the CSCC of $z$.

*Proof* Assume that $z$ is an ASCC. Then $z$ is not reachable from $R_1(z, y)$. Hence $R^*(R_1(z, y), y)$ and $R^*(x, z)$ cannot have a point in common, implying $C = \emptyset$. Now, assume $z$ belongs to some CSCC $S$. Then, all states in $S$ are forward reachable from $R_1(z, y)$ and backward reachable from $z$. Hence, $CSCC_z \subseteq C_z$. Conversely, all states in $C$ are reached by $z$ and can reach $z$. Thus, $C_z \subseteq CSCC_z$. Hence, $CSCC_z = C_z$ (QED Lemma).

#### *2.2.2 Finding A Shortest Path Between Two Sets*

Let $Src \subseteq V$ and $Dest \subseteq V$ be two subsets of vertices. We want to find a shortest path from $Src$ to $Dest$. The following algorithm returns a path $(s_0, \ldots, s_n)$, where $s_0 \in Src$ and $s_n \in Dest$, and $n$ is minimum. The algorithm works in two stages. In the first stage, using breadth-first search, all states distance $1, 2, \ldots$ from $Src$ are generated until $Dest$ is reached. This search mechanism guarantees that the path length is shortest. During the second pass, one such shortest path from $Src$ to $Dest$ is found, by searching backward from $Dest$ until $Src$ is reached.

***Finding A Shortest Path***

**1 Let** $i = 0$, $F_0 = src$.

**2 While** $Des$ **not reached** ($F_i \cap Dest = \emptyset$),

  **2.1** $F_{i+1} = R_1(F_i, y)$.

  **2.2** $i = i + 1$.

**3 Let** $n = i$ **and** $s_n \in F_n \cap Dest$.

**4 For** $j = n - 1$ **down to 0,**

  **4.1** $B_j = R_1(x, s_{j+1})$.

  **4.2 Let** $s_j \in F_j \cap B_j$.

**5 Return** $(s_0, s_1, \ldots, s_n)$.

This algorithm is applied in several places in the debugger, and is a useful utility. The following figure depicts how the algorithm works.

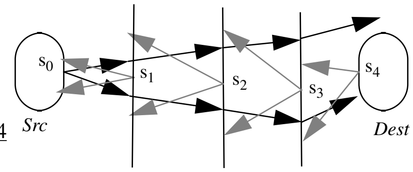



Figure 4

During the forward search, the set of reachable states are computed until *Dest* is reached. Then, the backward search starts by picking a state in the intersection of reachable states and *Dest*. One backward reachability step is performed, the intersection with reachable states in that level is taken, and an arbitrary state is chosen. The process continues until *Src* is reached.

## 3 Debugging Algorithms

In this section, we present BDD-based algorithms for finding an error trace. The first BDD-based algorithm for this problem was presented in [Tou91] ([Ste92] implemented a debugger based on ideas of [Tou91]). This algorithm depended on having the transitive closure of system. However, as [Hoj92] points out the transitive closure computation is rather expensive. Here, we present another debugger which needs very little pre-processed information. Specifically, we only assume there are some bad states, and we are given a set $B$ containing all bad states. Recall that a state is bad if it is contained in some bad cycle. Note that the algorithms of [Hoj92] return a set which is larger than the set of bad states.

*Definition* A ***bad CSCC*** of $Q$ is one that is not completely contained in any one of its cycle sets.

*Lemma* A CSCC is bad iff it contains a bad cycle (see theorem 2).

The algorithm consists of two parts.

1. a bad CSCC closest to an initial state is found.
2. a short cycle in this bad CSCC is found heuristically.

### 3.1 Finding a Bad CSCC

The following algorithm finds a bad CSCC which has a shortest path from some initial state. The algorithm starts from the set of initial states, and computes the set of reachable states $\hat{R}$ until the set $B$ is reached. Then, it arbitrarily chooses a state $x$ in $\hat{R} \wedge B$. If $CSCC_x = \emptyset$ or if $CSCC_x$ is not bad, the vertices in $SCC_x$ are deleted from $B$. Call this new set $\hat{B}$. Another $x \in \hat{R} \wedge \hat{B}$ is arbitrarily chosen, and its CSCC is computed. This process of choosing a state, examining its CSCC, and deleting the states in its SCC if the CSCC is not bad, continues until a bad CSCC is found or $\hat{R} \cap \hat{B} = \emptyset$. In the latter case, more reachability steps are performed until $\hat{R} \cap \hat{B}$ is non-empty again, in which case we fall back into the loop of choosing a state arbitrarily and examining its CSCC.

***Finding A Bad CSCC***

**1 Let** $\hat{R} = I$ **and** $\hat{B} = B$.

**2 While a bad CSCC has not been found**

  **2.1 While** $\hat{R} \cap \hat{B} \neq \emptyset$

    **2.1.1 Let** $x \in \hat{R} \cap \hat{B}$.

    **2.1.2 If** $CSCC_x$ **is bad, return** $CSCC_x$.

      **else** $\hat{B}(x) = \hat{B} \cap \overline{SCC_x}$.

**2.2 Let $\hat{R} = R_1(\hat{R}, y)$, and go back to 2.1.**

*Lemma* The above algorithm reports a bad cycle closest to the initial states.

*Proof* Let $x$ be the first state reported by the above algorithm such that $CSCC_x$ is bad. We know there is a bad cycle in $CSCC_x$, containing $x$. Call this cycle $C$, and let $l$ denote the minimum distance from $C$ to the set of initial states $I$. $l$ is also the distance from $x$ to $I$ by construction. Let $C'$ be any other bad cycle. Let $y \in C'$ be a closest state in $C'$ to the initial states. Since $B$ contains all states involved in some bad cycle, $y \in B$. The distance between $y$ and the initial states cannot be less than $l$, since otherwise $y$ would have been reached before $x$, and $CSCC_y$ which is a bad CSCC would have been reported. It follows that $C$ is a closest bad cycle to the initial states (QED Lemma).

### 3.2.2 Finding A Bad Cycle In A Bad CSCC

Now that we have found a bad CSCC, the next step is to find a short bad cycle in this set. This problem is difficult, as illustrated by the following lemma.

*Lemma* Let $G$ be a graph, and $\gamma_1, \ldots, \gamma_n$ be subsets of $V$, the vertices of $G$. Then, answering the question of "whether there exists a cycle of length less than $k$ not contained in any of the $\gamma_i$'s" is NP-Complete.

*Proof* 1. Proof of membership in NP. Just guess the cycle, and check whether the cycle is contained in any of the $\gamma_i$'s. The check can clearly be done in poly-time.

2. Proof of NP-Hardness. We reduce "Min Cover" ([GJ79]) to our problem. Let a set $S = \{s_1, \ldots, s_m\}$, and subsets of $S$, $U_1, \ldots, U_n$ be given. Create a complete graph $G$ with $n$ vertices, $\hat{U}_1, \ldots, \hat{U}_n$. Create $m$ subsets of vertices of $G$ (cycle sets), $\hat{S}_1, \ldots, \hat{S}_m$, as follows: $s_i \notin U_j$ iff $\hat{U}_j \in \hat{S}_i$. Note that size of $G$ is polynomial in the size of the original min covering problem. Now assume cycle $C$ with vertices $\{\hat{U}_{i_1}, \ldots, \hat{U}_{i_k}\}$ of length $k$ is not contained in any of the $\hat{S}_i$'s. Therefore, for every $s_j \in S$, there is some $U_{i_j}$ which contains it. Hence, there is a cover of size $k$ for the set $S$. Conversely, if there is a cover of size $k$, there is a set of vertices in $G$, which is not contained in any of the $\hat{S}_i$'s. Since $G$ is complete, there is a cycle among these vertices (QED Lemma).

Let $S$ be a bad CSCC. The following algorithm finds a short bad cycle in $S$. It constructs a bad cycle gradually by extending an existing path. Let $P = (x_1, \ldots, x_m)$ be the current path. All the cycle sets not containing $P$ are marked as inactive. Let $\gamma_{i_1}, \ldots, \gamma_{i_k}$ be the current set of active cycle sets, ordered in some arbitrary manner (more experiments may show that some orderings give better results than others). If all cycle sets are inactive, we augment $P$ by a path from $x_n$ to $x_1$, and report this cycle since this is a cycle that is not contained in any cycle set. If there are some active cycle sets, we heuristically try to find a state which would cause many active cycle sets to become inactive. Let $l$ be the maximum number such that $S \not\subset (\bigcup_{x_i \in P} x_i) \cup (\bigcup_{j=1}^{l} \gamma_{i_j})$. Let $Q = (\bigcup_{x_i \in P} x_i) \cup (\bigcup_{j=1}^{l} \gamma_{i_j})$. We choose a state $x \in \bar{Q} \cap S$ such that the distance between $x_n$ (the last state of $P$) and $x$ is minimum. For this task, we use the general shortest path finding algorithm. Note, by augmenting $P$ by this path, we can mark $\gamma_{i_1}, \ldots, \gamma_{i_l}$ as inactive. The following algorithm implements this idea.

***Extending A Bad Path***

**1 Let $Q = \bigcup_{x_i \in P} x_i$ and $l = 1$.**

**2 While not done**

**2.1 If $l = k$ or $S \subseteq \theta \cup \gamma_{i_l}$, find $x \in \bar{Q} \cap S$ by calling the path finding algorithm, with last state of $P$ being the source and $\bar{Q} \cap S$ the destination.**

**2.2 Else, let $Q = Q \cup \gamma_{i_l}$. Mark off $\gamma_{i_l}$, let $l = l + 1$, and continue.**

**3 Return a shortest path from $x_m$ to $x$.**

Recall that the algorithm which finds a bad CSCC also gives us state $y$ of the cycle. The following algorithm uses the above path extension method and state $y$ to find a short bad cycle in $S$.

***Finding A Short Bad Cycle***

**1 Let $x_1 = y$ and $P = (x_1)$.**

**2 While not done**

**2.1 Mark as inactive all the cycle sets which don't contain $\bigcup_{x_i \in P} x_i$.**

**2.2 If there are no more active cycle sets, then find a path from the last state of $P$ to $x_1$, and quit.**

**2.3 Let $x_i$ be the last state of $P$. Extend $P$ by calling the path extension routine. Augment $P$ by this extension.**

*Remark* The path from the last state of current path to the state returned by the path extension routine may contain vertices which are not contained in the current active cycle sets. Step 2.1 of the algorithm ensures that the effects of such states are taken into account, and thus more cycle sets can become inactive.

To further decrease the cycle's length, the following algorithm can be used as a post-processing step.

***Decreasing Bad Cycle's Length***

1. **Start with all nodes in the cycle, and find a minimum cover, where a cover is a set of vertices such that for each cycle set, there exists a vertex not in that cycle set.**
2. **Arrange the points in the cover in the same order as the original bad cycle.**
3. **Find a shortest path between consecutive points of the cover.**

It is easy to see that the length of the new cycle is guaranteed to be no longer than the original length (note that the vertices in the cover are arranged in the same order as the original cycle). Step 3 of the above algorithm can be improved by forming a new covering problem after the shortest path between two points in the new cover is calculated; some vertices in the new cover may be eliminated since the vertices in the path between two points can cause some cycle sets to become inactive.

## 4 Early Failure Detection

In this section, we describe an enhancement of the technique for EFD in [Hoj92]. This attempts to avoid computing the full-reachability set. We also explain how our debugger works with EFD.

### 4.1 Enhanced Early Failure Detection

Let $\Gamma = \bigcup_j \gamma_j$. The bad cycles can be classified into three sets:

1) ***Cycles of the first kind*** which lie entirely in $\bar{\Gamma}$.

2) ***Cycles of the second kind*** which intersect both $\Gamma$ and $\bar{\Gamma}$.

3) Cycles of the third kind which are bad cycles completely contained in $\Gamma$.

It has been noted in practice many simple errors lead to bad cycles of the first or second kind. Techniques of [Hoj92] find these errors quickly. However, these techniques still first compute the full reachability set, when in practice many early errors can be computed with only a partial set of reachable states. In effect, as we compute the set of reachable states, we perform a cheap test, which may find easy 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 in $D$ 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 current set equal to $R$, go to step 1, continuing the BFS 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 in this set, by calling EFD detection. Since easy errors are almost always of the first kind, and since checking cycles of the first kind is less computationally expensive than checking cycles of the second kind, we only invoke EFD with cycles of first kind (one can however as easily check for cycles of the second kind as well). The process continues until an error is found, or the full set of reachable states is computed.

### 4.2 Debugging and Early Failure Detection

Our debugger can be used with enhanced EFD. Assume EFD finds bad cycles of the first kind, and return a set of states containing some cycles of the first kind. The debugger then finds a CSCC containing a cycle of the first kind, and subsequently a bad cycle in this CSCC.

In general, one can relax the condition stated in section 3 that the set $B$, which is the input to the debugger, must contain all bad states. All that is required is that $B$ contains some bad state. The debugger guarantees that a bad cycle in $B$ is chosen whose distance to the initial states in minimum.

## 5 Debugging for Fair CTL

[Hoj93] described a method for performing language containment using L-automata, and fair CTL model checking in a unified environment. Using these ideas, we describe algorithms for reporting errors for fair CTL formulas, which use the utilities presented in section 3. We first briefly describe the syntax and semantics of fair CTL, followed by debugging and EFD for fair CTL.

### 5.1 Fair CTL ([Eme87])

*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 $\bar{f}$, $f+g$, $EXf$, $E[fUg]$, 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 $L(s)$ are subsets of AP. We assume that every state in a Kripke structure has some successor. If for state *s*, $P \in L(s)$, we say that *P* is true at *s*. If $P \notin L(s)$, it is assumed that $\bar{P}$ is true there. A process *S* can be considered a Kripke structure over $\Sigma$ by ignoring the edge labels. The atomic propositions are now the outputs produced by a node.

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

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

We say $\phi$ holds for $S$ if it holds at some initial state. For ease of readability, the following syntactic abbreviations are used: $AXf \equiv \overline{EX\bar{f}}$, $EFf \equiv E[trueUf]$, $AFf \equiv \overline{EG\bar{f}}$, $AGf \equiv \overline{EF\bar{f}}$, $A[fUg] \equiv \overline{E[\bar{g}U\bar{f}\bar{g}]} \wedge \overline{EG\bar{g}}$.

*Definition* A CTL formula is in ***positive normal form*** (PNF) if all negations are applied to atomic propositions, and the only Boolean operators used are "AND" and "OR". Any CTL formula can be written in this form.

*Definition* A CTL formula is a ***for all CTL (ACTL) formula*** if when written in PNF it does not involve existential quantification.

One is usually interested in restricting the set of paths considered in determining the truth of a formula $\phi$ in a Kripke structure *S*. Hence, one restricts the set of paths of the system to the "fair" paths.

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

*Remark* Assume a Kripke structure *S*, a formula $\varphi$ of the type $F^{\infty}f$ (or $G^{\infty}f$) where $f$ is a CTL formula, and a path $\pi$ 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 $\pi$ iff $F^{\infty}Q$ (or $G^{\infty}Q$) is true of $\pi$. Hence, we assume that only atomic propositions are used in conjunction with $F^{\infty}$ and $G^{\infty}$.

*Definition* A ***general fairness constraint*** is any Boolean combination of $F^{\infty}$ and $G^{\infty}$. A ***canonical fairness constraint (CFC)*** 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 CFC's. In practice, usually a subset of the CFC's is supported, e.g. $\prod_{i=1}^{n}O^{\infty}P_i$, where $O^{\infty}$ is one of the two infinitary operators. Note that any CFC can be written in sum-of-product (***SOP***) form, i.e as the sum of a set of product terms of the from $\prod_{i=1}^{n}O^{\infty}P_i$ (this translation may be exponential). We will use this reformulation in the next section.

*Definition* A formula in the logic ***fair CTL*** is a pair $(\phi, \Phi)$, where $\phi$ is a CTL formula and $\Phi$ is a canonical fairness constraint. The truth of $(\phi, \Phi)$ is determined at a state $s$ by only considering the paths which satisfy the fairness constraint $\Phi$ (these paths are called ***fair paths***). A state from which there exists a fair path is called a ***fair state***. A cycle in a system $S$ satisfying the fairness constraint $\Phi$ is called a ***fair cycle***. A CSCC containing a fair cycle is called a ***fair CSCC***.

### 5.2 Fair CTL Debugging

Since the introduction of CTL model checking ([Cla96]), there have been several debuggers for CTL formulas ([Lon92]). None of these have been described in the literature. But, it appears that they only deal with ACTL formulas, with no fairness constraints or restricted fairness constraints. Here, we describe a debugger for fair CTL formulas based on the debugging environment of section 3 for LC.

For fair ACTL formulas, the debugger can automatically return an error trace, whereas for formulas involving existential quantification, user interaction is required. In all cases, the option to interact with the debugger is provided. As for optimality of the error trace, for fair ACTL formulas, we prove the problem of returning a short error trace is NP-complete, and offer a heuristic solution. However, if there are no fairness constraints, we are able to prove a minimality result.

Assume a fair CTL formula $(\phi, \Phi)$ is given, where $\phi$ is in PNF, and $\Phi$ is in SOP form. Note that the fairness constraint $\Phi$ is restricting the paths to be considered, and can be thought of as a restriction on our model. The formula $\phi$ does not hold for the system if it does not hold at some initial state. In this case, we have to provide a proof of why it does not hold in that initial state. The debugger works incrementally, giving debugging information one operator at a time. At every point, the input is a CTL formula $\psi$, and a fair state $x$ in which $\psi$ is false. The formula $\psi$ and state $x$ constitute the state of the debugger. Every state has a unique identifier, and the user can refer back to any previously visited state. This feature is used to backtrack and explore other paths to get further debugging information.

As an example, consider the formulas $AGAFp$. The debugger first gives a path from some initial state $s_0$ to some state $x$ where $AFp$ is false. Then, if the user wants to see a proof of why $AFp$ is false at $x$, the debugger presents a fair path staring at $x$ such that $p$ is false along the path. Note that a fair path involves an initial segment and a cycle, where the cycle satisfies all fairness constraints expressed by $\Phi$. To prove the problem of finding a short fair cycle in NP-complete, we need the following result ([Hoj93]).

*Theorem 3* Let $(\phi, \Phi)$ be a fair CTL formula on a system $S$, where $\Phi$ is in the product form $\prod_{i=1}^{n}O^{\infty}P_i$. Then:

a. there is a set of recur edges and cycle sets on $S$ such that bad cycles and paths are the same as fair cycles and paths, respectively;

b. given a system with only a set of cycle sets, there exists a fairness constraint of the form $\prod_{i=1}^{n}F^{\infty}P_i$, such that the bad and fair cycles are the same.

*Theorem 4* Let a graph $G$ with fairness constraints of the form $\prod_{i=1}^{n}F^{\infty}P_i$ be given. Finding a fair cycle of length less than $k$ in $G$ is NP-complete.

*Proof* Membership in NP is clear: just guess the cycle and check for its fairness. Also, by the above theorem, we have a polynomial reduction from a problem proved NP-complete in section 3 to this problem (QED theorem).

*Corollary* Given a system with CFC $\Phi$, finding a fair cycle of length less than $k$ is NP-complete.

We now present the algorithm for fair CTL debugging. Note that by calling the LC, we use the heuristics for finding a short bad cycle to find a short fair cycle. The algorithms proceeds as follows depending on $\psi$.

1. $\psi = P$, for some atomic proposition $P$. In this case, we report $x$ as an error, and ask the user whether a fair path from $x$ is desired. Note that the user may have thought $x$ is not fair, and wants a proof of its fairness. To find a fair path, we consider the CFC $\Phi$ in SOP form. Since $x$ is a fair state, there must exists a path from $x$ which satisfies one of the product terms in $\Phi$. Using theorem 3, we create an LC problem for each product term. We call the debugger of the last section with initial state $x$ and the set of bad states being the fair states. At least, one of these calls to LC for a given product term will return a fair path which is (heuristically) minimized.

If there are fair paths with respect to only one of the product terms, then we are guaranteed that the path from $x$ to the reported fair cycle is minimum. Otherwise, the first product term for which there is a fair path is chosen. It is possible that there is another product term and a fair path for this term, such that the path from $x$ to the fair cycle is shorter.

2. $\psi = f+g$, $f$ and $g$ CTL formulas. In this case, we ask the user which of the formulas to pursue further, since both are false. Assume $f$ is chosen. The debugger is recursively called with $f$ and $x$. The user can come back to this state,

and call the debugger with $g$ and $x$ later on.

3. $\psi = f \wedge g$. Choose (arbitrarily if there is a choice) one of the formulas $f$ or $g$ which does not hold in $x$, and call the debugger with this formula and $x$.

4. $\psi = AXf$. Choose (arbitrarily if there is a choice) a fair next state of $x$ in which $f$ does not hold. Call the debugger recursively with $f$ and $x$.

5. $\psi = AGf$. Call the shortest path algorithm of section 3 to find a shortest path between $x$ and the set of fair states satisfying $\bar{f}$. Let $y$ be a closest fair-state to $x$ in which $f$ does not hold. Report the path from $x$ to $y$, and call the debugger recursively on $f$ and $y$.

6. $\psi = AFf$. In this case, we have to find a fair path from $x$ along which $f$ never becomes true. The algorithm is similar to that for the case of $\psi = P$. Let $S = \bar{f} \cap \text{fair-states}$. For each product term in the SOP of CFC, call the LC debugger with $S$ as the bad set and $x$ as the initial state. When the error trace is found, the path is printed, and the user can ask why $f$ does not hold in any state along the path.

7. $\psi = A[fUg]$, where $f \neq true$. Let $Q$ be the set of fair states satisfying $g$. If $Q = \varnothing$, report it to the user, and ask whether a fair path is desired. Otherwise, let $\bar{P}$ be the set of states satisfying $\bar{f} \wedge \bar{g}$ which can reach $Q$ and are reached by $x$. Note that $\bar{P}$ is easily computed using reachability steps. An error trace consists of a path from $x$ to some $y \in \bar{P}$ such that $f$ holds along all states of the path except for $y$, and a path from $y$ to some $z \in Q$. To find the shortest such path, we proceed as follows for $n = 0, 1, 2, \ldots$. The set of states in $\bar{P}$, reachable in exactly $n$ steps from $x$ is computed. Call this set $\bar{P}_n$. Then, the length of the shortest path from $\bar{P}_n$ to $Q$ is computed. Let it be $m$. It is then recorded that an error trace of length $n + m$ exists. Let $\mu$ denote the distance between $\bar{P}$ and $Q$. The process continues until $n + \mu$ is not smaller than the minimum recorded error trace length. The algorithm then constructs an error trace of minimum length. The error trace is printed, and the user is given the following choices:
1. a proof of why $f$ or $g$ are $true$ or $false$ along the path,
2. completion of the path by a fair cycle.

8. $\psi = EXf$. Tell the user how many next states $x$ has. Ask for user constraints on the set of states of interest. The constraints are a set of state variables and values they must hold. Recall that each state is a product state. Enumerate those next states which satisfy the user constraints until the user chooses a next state $y$ to follow. Call the debugger recursively with $f$ and $y$.

9. $\psi = EGf$. If $f$ does not hold in $x$ the debugger with $f$ on $x$. Otherwise, let $N(x)$ denote the set of next states of $x$, subject to some user constraints. Let $S_1 = N(x) \cap f \cap \text{fair-states}$. Let $S_2 = N(x) \cap \bar{f} \cap \text{fair-states}$. Ask the user to choose $S_1$ or $S_2$. If $S_1$ is chosen, enumerate all states in $S_1$ until the user chooses a next state $y$. In this case, the user is following the path expected to have satisfied $EGf$. Call the debugger recursively on $EGf$ and $y$. If $S_2$ is chosen, do the same except call the debugger recursively on $f$ and $y$. In this case, the user had thought $f$ holds at $y$, whereas it does not.

10. $\psi = EFf$. In this case, we know for all next states of $x$, $\psi$ does not hold. Let $N(x)$ denote the set of next states of $x$, subject to user constraints. Enumerate $N$ until the user chooses a next state $y$. Ask the user to choose between proving why $EFf$ or $f$ does not hold in $y$. Call the debugger recursively with the chosen formula and $y$.

11. $\psi = E[fUg]$, where $f \neq true$. If $x$ satisfies $\bar{f} \wedge \bar{g}$, print x, and ask whether the user wants to prove why $\bar{f}$ or $\bar{g}$ is not $true$ at $x$. Otherwise, ask the user if a proof of why $f$ or $\bar{g}$ holds at $x$ is desired. If not, tell the user the number of next states satisfying $f$, $\bar{f}$, and $\bar{f} \wedge \bar{g}$. Ask, which one the user want to pursue; enumerate the corresponding set of states until the user chooses a next state $y$. Call the debugger with $E[fUg]$ and $y$.

*Remark* In some cases, such as $\psi = A[fUg]$, we may have to prove why $f$ holds at some state $x$. Our debugging paradigm, however, is that we are at some state $x$, where some formula $\psi$ does not hold. To use this paradigm, we say that at $x$, $\bar{f}$ does not hold. We need to compute $\bar{f}$ in PNF given $f$ in PNF. The important point is that we do not need to compute any new BDD's satisfying the intermediate formulas. For instance, assume $f = p + q$. Then, $\bar{f} = \bar{p} \wedge \bar{q}$. The intermediate formulas are $\bar{f}$, $\bar{p}$, and $\bar{q}$. Since, we already have BDD's representing $f$, $p$, and $q$, and since complementation in BDD's is a change of pointers, we do not need any new computations.

*Remark* With formulas involving existential quantification, it may be that there is some unfair path along which the formula is valid. The user may want to explore this path. Hence, if the formula is not an ACTL formula, the first step of debugging is to run the model checker on the system without any fairness constraints. Then at every step where the formula is an existential formula, we report the unfair next states. If the user chooses one of these states, the debugger will only explore unfair reachable states, and ignore the fairness constraints.

*Lemma* Let $\phi$ be an ACTL formula. Assume no fairness constraints are given for the system $S$. Assume $\phi$ does not hold for $S$. Then, the following is true of the error trace $\pi$ returned by the debugger, which consists of an initial path and a final cycle:

1. The length of the initial path is minimum among all error traces.

2. Let $s$ be the last state of the initial path. The cycle returned is a shortest cycle among cycles containing $s$.

*Proof* Part 1 follows from the algorithm, using induction on the definition of ACTL formulas. Part 2 follows, since we use a shortest path algorithm to find a path from $s$ to itself (QED Lemma).

### 5.3 Early Failure Detection (EFD) in Fair CTL

So far, we do not know how to extend EFD techniques to general CTL formulas. However, the technique described in section 4 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 $\phi$ be an ACTL formula in PNF, interpreted over CFC $\Phi$. Let $s \in S$. Then, if $\phi$ does not hold at $s$ (over $\Phi$) in $S$, then $\phi$

does not hold at $s$ (over $\Phi$) in $R$.

*Proof* We use induction over the definition of CTL formulas. Assume $\phi$ does not hold at $s$ in $S$. Note that $\phi$ 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 are 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 $s$, $\phi$ 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 $\pi$, at which $f$ does not hold. By inductive assumption, $f$ does not hold of $x$ in $R$. Since $\pi$ is a fair path in $R$, $\phi$ does not hold of $s$ in $R$

5. $AXf$. Proof is similar to case 4.

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

*Remark* If the formula $\phi$ is not in PNF, then the above lemma is not true, since complementation and universal quantification can be used to create existential quantification.

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 L-automata. The problem of extending quick failure checks ([Hoj92]) to ACTL formulas remains open.

## 6 Comments On Experiments

We have implemented the LC debugger of section 3, with the exception of the post-processing step to decrease the bad cycle's length. The extended early failure detection algorithms for LC and fair CTL have also been implemented. The algorithm for fair CTL debugging is not yet implemented.

We have used the LC debugger on some simple examples for property checking and timing verification ([Bal92]). Its running time is small compared to the running time of the LC algorithms and it produced small error traces on our examples. However, we have not tested the heuristics for finding short bad cycles on a larger set of more complicated examples.

We tested the extended EFD algorithms on the encyclopedia version of the dinning philosophers example. In this version, a philosopher can be in four states: reading, hungry, eating, and thinking. To prevent a philosopher from eating forever, a fairness constraint is added which requires the system to exit the eating state infinitely often. To test our algorithms, we removed this fairness constraint, and checked for non-starvation. This check fails, since a philosopher can now eat forever, and hence prevent others from eating. In the following tables, we report the CPU times for LC, LC with extended EFD, MC, and MC with EFD. In all cases, we report how long the algorithms spent doing reachability analysis, and how long they spent in checking the property.

| | EFD | | No EFD | |
|---|---|---|---|---|
| | Reach | Check | Reach | Check |
| LC | .30 | 1.04 | 61.60 | 6.61 |

| | EFD | | No EFD | |
|---|---|---|---|---|
| | Reach | Check | Reach | Check |
| MC | .30 | 70.70 | 46.30 | 239.0 |

Table 1

This table show the CPU times for checking starvation on the dinning philosophers example. The upper and lower halves contain the times for LC and fair CTL MC respectively. Running times with and without EFD are indicated in each case. The CPU times are measured in seconds, on a DEC-station 5000.

It should be noted that the large difference in running times between LC and MC in property checking is partly due to the fact that the full computation is avoided in the case of LC, since the EFD techniques of [Hoj92] find a cycle of the first kind early in the computation. It is possible that these techniques can be extended to MC as well.

## 7 Conclusion

We presented BDD-based debugging techniques (which can deal with large graphs) for language containment using L-automata and fair CTL model checking. Our debugger for LC has the following advantages. First, it requires very little information to compute a bad error trace; in general, it requires a set which includes some bad state. Hence, the debugger is able to work with the EFD algorithms of [Hoj92]. Second, if the input to the debugger contains all bad states, it returns an error trace whose path to the bad cycle is minimum. This is augmented with a short bad cycle using heuristics (we proved that the problem of finding a minimum length bad cycle in NP-complete).

The debugger for fair CTL is based on the LC debugger, using results of [Hoj93]. This debugger is interactive, in order to allow the user to explore many alternatives.

We have also extended the techniques of [Hoj92] for EFD so that in some cases the full reachability computation can be avoided. This method of avoiding the full reachability set applies to LC and to ACTL formulas.

## References


[Agg83] S. Aggarwal, R. P. Kurshan, K. Sabnani, "*A Calculus for Protocol Specification and Validation*", in Protocol Specification, Testing and Verification III (193) North-Holland, pp. 19-34.

[Alu92] R. Alur, A. Itai, R. P. Kurshan, "*Timing Verification by Successive Approximation*", Computer-Aided Verification, 1992.

[Bal92] Felice Balarin, A. S. Vincentelli, "*A Verification Strategy for Timing-Constrained Systems*", Fourth Workshop On Computer-Aided Verification, 1992.

[Cla86] E. M. Clarke, E. A. Emerson, 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.

[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.

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

[Hoj93] Ramin Hojati, Thomas Shiple, Robert Brayton, Robert Kurshan, "*A Unified Environment for Language Containment and Fair CTL Model Checking*", submitted to Design Automation Conference, 1993.

[GJ79] R. Garey, D. S. Johnson, "*Computers and Intractability, A Guide to the Theory of NP-Completeness*", W. H. Freeman and Company, 1979.

[Kur90] R. Kurshan. "*Analysis of Discrete Event Coordination*", Lecture Notes in Computer Science, 1990.

[Lon92] D. Long, private communication.

[Ste92] Paul Stephan, Huey Wang, private communication.

[Tou90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, A. S. Vincentelli, "*Implicit State Enumeration of Finite State Machines Using BDD's*", International Conference on Computer-Aided Design, 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.