Efficient ω-Regular Language Containment

Ramin Hojati (UC Berkeley)[1]
Herve Touati (DEC PRL)
Robert P. Kurshan (AT&T Bell Laboratories)
Robert K. Brayton (UC Berkeley)

Abstract

One method for proving properties about a design is by using L-automata [Kur90]. The main computation involves building the product machine of the system and specification, and then checking for cycles not contained in any of the cycle sets (these are sets of states specified by the user). In [Tou91] two methods were introduced for performing the above task; one involves computing the transitive closure of the product machine, and the other is an application of a method due to Emerson-Lei ([Eme86]). We have implemented both methods and extended them. We introduce a few general-purpose operators on graphs and use them to construct efficient algorithms for the above task. Fast special checks are applied to find bad cycles early on. Initial experimental results are encouraging and are presented here.

1 Introduction

Implementation verification involves checking whether two different representations of a system are equivalent. An example is checking whether a logic implementation faithfully implements a register-transfer language description. ***Design verification*** is the process of verifying whether a system has a set of desired properties. An example is checking that a communication protocol does not fall in a deadlock state. Presently, design verification is done by extensive simulation.

Design verification is the more challenging and important problem. Two general approaches using formal verification exist. The first employs general theorem-proving techniques to prove a result about some aspect of the design. Verification based on *Boyer-Moore theorem prover* or *HOL verification system* are examples. The second approach uses specialized logics or automata on infinite strings (***w-automata***) to express properties about a set of interacting finite state machines which model the design. Examples are *Computation Tree Logic* ([Cla86]), process calculi ([Bou89]) and *L-automata* ([Kur90]). This work is concerned with the use of L-automata in formal design verification.

1.1 The L-automata Environment

Specification of both systems and properties in this environment is done by the use of ω-automata. [Ch74] provides an introduction to the subject. Here, we briefly cover a few relevant extensions made to the basic theory. For a more detailed explanation of these, see [Kur90]. The system in this environment is modeled by a set of L-processes, which are similar to Moore machines. An ***L-process*** consists of 6 components.

1) States. The set of states of the L-process.

2) Transition matrix. Specifies the state transitions of the machine. The entries of this matrix are boolean equations, specifying the conditions under which a transition is taken.

3) Initial states. The set of initial states of the machine.

1. During this work, the first author was supported by an SRC grant, under contract number 91-DC-008.

4) Output function. A function of states which specifies a set of outputs for each state. At a given state, each machine chooses one of its outputs non-deterministically. Since the systems are modeled as closed systems, outputs and inputs are the same; they are collectively called ***selections***.

5) ***Recur edges***. A set of edges in the machine with the interpretation that if the machine follows a recur edge infinitely often, then the resulting sequence is rejected.

6) ***Cycle sets***. A set of sets of states with the interpretation that a sequence of states is rejected if the set of states traversed infinitely often is contained in one of the cycle sets.

A string is accepted if one of its runs is not rejected, where a ***run*** of a string is a path in an L-process which results in that string. The model of concurrent computing used in this environment is know as the ***selection/resolution model***, and consists of two basic steps which are repeated indefinitely. During ***selection***, all machines simultaneously and non-deterministically choose one of the outputs possible for their respective states. During ***resolution***, each machine chooses a new state based on both its current states and the ***global selection***, i.e the set of outputs produced by all machines during the previous selection step.

L-automata, which are used for specifying properties or ***tasks***, are similar to L-processes with two differences; first L-automata have no outputs, second is the way recur edges and cycle sets are interpreted. In the case of L-automata, a string is accepted either if for one of its runs some recur edge is traversed infinitely often or the infinite portion of the string is contained in some cycle set. Note that this is complementary to the acceptance condition of L-processes.

Verification now consists of modeling a system and its environment as a closed system of L-processes. Non-determinism is used heavily to express concisely all possible behaviors. A property is then represented using an L-automata, which takes inputs from the selections of the L-processes. A complex property is usually broken down into several smaller properties, each one represented by a ***deterministic L-automaton***, i.e. an L-automaton which has deterministic transitions but may have multiple initial states. In our environment, all L-automata are deterministic. Verifying whether a system has a property is then reduced to checking whether the language accepted by the system is contained in the language of the L-automata of the property ([Kur90]). The language of the system is the language accepted by the automaton obtained as the product of all of the L-processes with outputs ignored. This acceptance check is sometimes referred to as an ω-regular language containment check.

1.2 Product Machines

At two points in the verification process, we need to form product machines; first, to represent the system of L-processes by their tensor product. Second, to verify that a system has a property, we form the product machine of the L-processes and the L-automaton, and verify that all cycles of this product machine either contain a recur edge or are completely contained in some cycle set. The two products can be performed in one step.

A product machine is formed by taking the tensor product of the transition matrices, taking the Cartesian product of the initial states, and taking the union of the cycle sets and recur edges. Note that the last two unions are in terms of the product machines, i.e. a set of states of a product machine is a cycle set if and only if the set restricted to some component is a cycle set for that component. The case of recur edges is similar.

1.3 Fixpoint Computations

Let Q be a k-ary predicate over $D_1,\ldots, D_k$, where all D_i's are finite. Let $F(X)$ be a ***k-ary predicate transformer***, i.e. a unary function whose first argument is a k-ary pred-

icate and which returns a k-ary predicate. Assume F is ***monotone decreasing*** (or ***monotone increasing***), i.e. $(\forall Q(F(Q) \subseteq Q))$ $((\forall Q(Q \subseteq F(Q))))$. A ***fixpoint*** of F is a predicate Q such that $F(Q) = Q$.

Definition Let F be a k-ary predicate transformer. Define $F^i(X)$ by $F^i(X) = (F(F(...F(X))))$, where F is applied i times to X.

Definition Let F be a k-ary predicate transformer, which is monotone increasing. Define the ***greatest fixpoint of F given Q***, denoted by $\mu(X, Q).FX$, where Q is an k-ary monotone increasing predicate over $D_1,..., D_k$ by the set $F^i(Q)$ such that $F(F^i(Q)) = F^i(Q)$. Similarly, define the ***least fixpoint*** of a monotone decreasing k-ary predicate transformer F given Q, $\nu(X, Q).FX$ by the set $F^i(Q)$ such that $F(F^i(Q)) = F^i(Q)$.

Example Let $T(x, y)$ be the transition function for an unlabeled graph. Let $A(x)$ denote the set of initial states. The fixpoint computation $R^*(A, x) = \mu(X, A).(X(y) \vee \exists x(X(x) \wedge T(x, y)))$ computes the set of all the states reachable from A.

The computation $R^*(A, y) = \mu(X, A).(A(y) \vee \exists x(X(x) \wedge T(x, y)))$ is an alternate way to compute the set of reachable states. Note that the two methods compute the same set. However, they perform it differently. Similarly, the fixpoint computation $R^*(x, A) = \mu(X, A).(X(x) \vee \exists y(X(y) \wedge T(x, y)))$ computes the set of states which can ultimately reach A.

1.4 Our Contribution

The software COSPAN, described in [Har90] performs the language containment check by explicitly building the product machine and then examining the strongly connected components of an altered product machine, where the recur edges removed. If each strongly connected component of this altered machine is contained in a cycle set of the product machine, the check passes. In the case of failure, an error tracing mechanism interacts with the user to help locate the source of error.

[Tou91] presented a method based on Binary Decision Diagrams (BDDs) where explicit enumeration of the states of the product machine is not required. Two algorithms using BDDs were suggested in [Tou91]; one involving a transitive closure computation and one based on an application of the Emerson-Lei method introduced in [Eme86]. After implementing the transitive closure algorithm and noticing that this computation is usually very expensive, we proposed several algorithms based on ideas borrowed from the Emerson-Lei method. Checks for finding simple errors early on were also added. All of our algorithms have been implemented and are compared to each other on a limited set of examples.

The organization of the paper is as follows. Section 2 describes the algorithms for language containment check. Section 3 gives the experimental results. Section 4 defines complexity classes for BDD's. Section 5 is the conclusion. Due to lack of space, most proofs have been omitted.

2 Language Containment Check

In this section, we study several algorithms for checking language containment. In what follows, $P_1,..., P_n$ are a set of L-processes, where $P_i = (Q_i, T_i, I_i, O_i, R_i, C_i)$, and $A = (Q_A, T_A, I_A, O_A, R_A, C_A)$ is an L-automaton, defining a property of the system. Let $P = (Q, T, I, R, C)$ be the product machine $P = P_1 \times ... \times P_n \times A$, where the out-

puts are ignored. For all the algorithms which follow, we assume that all L-processes, L-automata, and product machines are represented by BDD's. Let γ_j for $j = 1, \ldots, n$ be the set of cycle sets of P. The language containment check is as follows. All of the algorithms we present, follow the same basic methodology.

Language Containment Algorithm

1) Let $P = P_1 \times \ldots \times P_n \times A$.

2) Let Q be the underlying graph of reachable states of P with the recur edges removed.

3) If all cycles (or equivalently cyclic strongly connected components) of Q are contained in some cycle set, then the check has passed. Otherwise, it has failed.

Definition A ***bad cycle*** is a cycle of the product machine $P = P_1 \times \ldots \times P_n \times A$, not contained in any of the cycle sets and not containing any recur edge. Equivalently, let Q be P, with recur edges removed from its transition function. Then, a bad cycle in P is a cycle in Q, not contained in any of the cycle sets.

The first algorithm we study involves computing the transitive closure of P. This algorithm is rather expensive because computing the transitive closure of a graph is generally an expensive operation. Iterated squaring, as a possible candidate to speed up the transitive closure operation, is described. Then, we describe three algorithms based on ideas borrowed from [Eme86] and [Tou91]. Methods for finding simple bad cycles early are presented afterwards, which complete all the details needed to present the final algorithm.

2.1 Algorithm Based on Transitive Closure

The first BDD based algorithm for this task appeared in [Tou91]. The algorithm uses transitive closure to represent strongly connected components. Let G be an unlabeled graph, represented by its transition function $T(x, y)$. Let $C(x, y)$ denote the transitive closure of G.

Definition Let G be as above. Define S such that $S(x, y) = 1$ iff x and y are in the same cyclic strongly connected component of G.

Lemma Let G, S, and C be as defined above. Then, $S(x, y) = C(x, y) \wedge C(y, x)$.

Definition Let $P = P_1 \times \ldots \times P_n \times A$. Define by $B(x)$ the set of all states of P, which are involved in some bad cycle.

The algorithm which follows, uses the transitive closure of Q. For each cycle set, the SCC's of Q, not contained in that cycle set are determined. If the intersection of all such sets is empty, the check passes. Otherwise it fails.

Language Containment Check Using Transitive Closure

1) Build the product of the component machines and the task. Let $T(x, i, y, r)$ represent the transition function of the product machine.

2) Remove the selection variables from the transition function, $T(x, y, r) = \exists i T(x, i, y, r)$.

3) Compute the set of reachable states, $R^*(I, y) = \mu(X, I).(X(y) \vee \exists x(X(x) \wedge T(x, y, r)))$.

4) Remove the unreachable states and the recur edges from the transition function.

$$T(x, y, r) = T(x, y, r) \wedge R^*(x) \wedge R^*(y)$$

$$T(x, y) = T(x, y, 0)$$

5) Compute the transitive closure of the transition function of the product

machine.

$$C(x,y) = \mu(X,T).(X(x,y) \vee \exists z(X(x,z) \wedge T(z,y)))$$

6) Let γ_j denote the j-th cycle set of the product machine. Then,

$$\hat{B}(x) = \bigcap_j \exists y(\overline{\gamma_j(y)} \wedge C(x,y) \wedge C(y,x))$$

7) In the case of failure ($\hat{B} \neq \varnothing$), call the debugger.

Theorem Let $\hat{B}(x)$ be the set calculated by the above algorithm. Then, $\hat{B}(x) = B(x)$.

Building the transitive closure of the product machine has proved to be expensive. In the next section, we discuss algorithms which can speed up this operation.

2.2 Iterated Squaring

In this section, we introduce a technique which can speed up some fixpoint computations on some graphs. Specifically this technique is useful on graphs with long chains of states, such as counters. Consider a fixpoint computation such as computing the set of reachable states of a graph. The usual method of computing this set can take $o(n)$ iterations, where n is the level of the underlying graph. Such computations will be denoted as ***linear computations***, where the worst case complexity is $o(n)$, n being the number of vertices of the graph. A method which reduces the time for such computations from $o(n)$ to $O(\log n)$ is ***iterated squaring***. [Bur90] introduced one iterated squaring technique for symbolic verification of μ-calculus. We describe iterated squaring techniques for two problems, namely computing reachable states and transitive closure of a graph G. For the first problem, we introduce one iterated squaring method, whereas for the second, we introduce two iterated squaring techniques which perform differently in practice.

Definition Let $T(x,y)$ denote the transition relation of a graph G. Let $T_k(x,y)$ denote the transition relation of a graph where there is an edge between x and y iff there is a path of length exactly k between x and y. Let $C(x,y)$ denote the transitive closure of G. Let $C_k(x,y)$ be the transition relation of a graph, where there is an edge between x and y iff there is a path of length k or less between x and y. Note that $C_n(x,y) = C(x,y)$, where n is the diameter of G.

2.2.1 Computing Reachable States

Recall that the linear computation of reachable states is accomplished by the fixpoint calculation., $R^*(A,y) = \mu(X,A).(X(y) \vee \exists x(X(x) \wedge T(x,y)))$. An iterated squaring form of the above computation is $R^*(A,y) = \mu(X,A).(X(y) \vee \exists x(X(x) \wedge T_{2^{k-1}}(x,y)))$. In order to calculate $T_{2^k}(x,y)$, one has to perform an expensive computation where three sets of variables (x, y, and z) are active at the same time, namely, $T_{2^k}(x,y) = \exists z(T_{2^{k-1}}(x,z) \wedge T_{2^{k-1}}(z,y))$. The above computation is usually time consuming, since the BDD corresponding to $T_{2^{k-1}}(x,y)$ or intermediate BDD's needed to build $T_{2^{k-1}}(x,y)$ can become large (as an aside, note that the number of edges represented by $T_{2^{k-1}}(x,y)$ should be of the same order as those represented by $T(x,y)$). As a rule, computations of the form $\exists z(A(x,z)B(z,y))$, which involve three sets of active variables take much longer than image computations, which involve two sets of active variables.

Our experiments indicate that the use of iterated squaring, as presented above, for computing the reachable states is not recommended. The experimental results were obtained running the dining philosopher examples. With only 6 philosophers, computing the reachable states using iterated squaring takes 60 seconds, where it only takes 1 second using linear computations. Hence, we did not perform further experiments using iterated squaring for computing reached states.

2.2.2 Computing Transitive Closure

Recall the linear computation for transitive closure of a graph G, i.e. $C(x,y) = \mu(X,T).(X(x,y) \vee \exists z(X(x,z) \wedge T(z,y)))$. We discuss two general methods of using the iterated squaring technique for this computation. Both of these methods compute $C_{2^k}(x,y)$ successively until a fixpoint is reached.

Iterated Squaring by Shifting We calculate $C_{2^k}(x,y)$ by calculating $C_{2^{k-1}}(x,y)$, then shifting it by $T_{2^{k-1}}(x,y)$ and OR-ing it with $C_{2^{k-1}}(x,y)$ (recall from last section how $T_{2^{k-1}}(x,y)$ is computed). The following describes the fixpoint computation $C(x,y) = \mu(X,T).(X(x,y) \vee \exists z(X(x,z) \wedge T_{2^{k-1}}(z,y)))$.

Note that by the above computation every path is added in only once. For example, assume $C_4(x,y)$ and $T_4(x,y)$ have been computed. Let a computation of the form $\exists x(A \wedge B)$ be called a ***path extension***. Then, the above path extension adds to C_4 only paths of length 5-8, i.e. an edge is added between x and y iff there is a path of length 5-8 between x and y in the original graph. For example, let *(x1, x2, x3, x4, x5, x6, x7)* be a path of length 6 in the original graph. Then, there is an edge *(x1, x3)* in $C_4(x,y)$, and an edge *(x3, x7)* in $T_4(x,y)$. So *(x1, x7)* is included in $\exists z(C_4(x,z) \wedge T_4(z,y))$. However, there are no edges for *(x1, x3)* and *(x3, x7)* in this path extension. Note that the edge *(x1, x7)* is added only once. We will see that in iterated squaring by folding such an edge can be added many times.

The number of iterations of this method is $\log n$, where n is the diameter of the graph. However, we need to perform two sets of path extensions, each with three sets of active variables at every iteration. On the other hand, we only need to save two functions at every step, namely the current $T_{2^k}(x,y)$ and $C_{2^k}(x,y)$. Based on our experiments, iterated squaring by shifting compared to linear computations, consumes more memory but has about the same running-time.

Iterated Squaring by Folding To calculate $C_{2^k}(x,y)$, we use the fixpoint computation, $C(x,y) = \mu(X,T).(T(x,y) \vee \exists z(X(x,z) \wedge X(z,y)))$. As an example, assume $C_4(x,y)$ has been calculated. After the above path extension, we get $C_{2-8}(x,y)$, where there is an edge between x and y iff there is a path of length 2-8 between them. To get $C_8(x,y)$, we add $T(x,y)$ to the result. Note that by this computation, an edge for a path may be added many times. For instance, the path *(x1, x2, x3, x4, x5, x6, x7)* creates the edge *(x1, x7)* three times: first, by *(x1, x3)* and *(x3, x7)*; second, by *(x1, x4)* and *(x4, x7)*; and third, by *(x1, x5)* and *(x5, x7)*.

Again, the number of iterations is $\log n$. This computation was faster than the linear computation in the experiments we performed. A main reason is that there is only one path extension with three sets of active variables. It appears that this method is more space consuming compared to linear computations. In the experimental section, we describe the results of our experiments with the transitive closure algorithm for doing language containment, where the transitive closure check was done using iterated squaring. In the next few sections, we describe algorithms which don't need the

transitive closure computation.

2.3 The Emerson-Lei Method and Modifications

In this section, we introduce four algorithms for the language containment check. The main idea for these algorithms is a computation introduced in [Eme86] for propositional μ-calculus, and adapted to check language containment by [Tou91]. We first introduce some operators on graphs, which are needed later.

Historical Remark The work in [Eme86] described a method for translating a subset of CTL* ([Cla86]) formulas into a subset of propositional μ-calculus, namely L_{μ_2}, for which polynomial-time algorithms are available. The important point about the model checking algorithm for L_{μ_2} is that it does not involve any transitive closure computation. The works described in [Eme87] and [Cla90] described methods to perform the language containment check for ω-automata using this polynomial subset of CTL*. [Tou91] formulated the language containment check for L-automata in L_{μ_2}. This is the first algorithm we describe. Our contribution was to enhance this algorithm. We view the Emerson-Lei method for performing language containment check as an operator trimming the state space. We introduce several new operators. Moreover, we describe a method for early failure detection.

2.3.1 Some Graph Operators

In what follows, let G be a graph, $V(x)$ the set of its vertices, $A(x) \subseteq V(x)$ a subset of its vertices, and $T(x, y)$ its transition function.

Definition Let G be a graph. A ***cyclic strongly connected component (CSCC)*** of G is a SCC of G which contains at least one cycle. Hence, all SCC's of G are CSCC's except for single node SCC's which don't have a self-loop, which are called ***acyclic strongly connected components (ASCC's).***

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 vertices 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 $S^*(x, A) = \nu(X, A).(S_1(x, X))$. Note these are computing the greatest fixed point contained in A. 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), 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).

Lemma If A contains a cycle γ, then $S^*(A, y)$ and $S^*(x, A)$ contain γ.

Definition We define one more one-step operator, the ***trim operator.***

$$Z_1(c, A) = \begin{cases} \bar{c} \wedge A & \text{if } (A \wedge R_1(c, y) \subseteq c) \text{ or } (A \wedge R_1(x, c) \subseteq c) \\ A & \text{otherwise} \end{cases}$$

where c and A are sets of states. This operator will be used on each cycle set to elim-

inate cycle sets γ_j that either have no exit from γ_j (***sink cycle sets***) or no incoming edges (***source cycle sets***). Recursive application of this operator, the ***recursive trim*** operator, defined by $Z^*(x,A) = \nu(Y,A).Z_1(\gamma_1, Z_1(\gamma_2, \ldots Z_1(\gamma_n, Y)\ldots))$, where A is some initial set of state, is also useful. The usefulness of recursive trim is because γ_j may be eliminated by Z^* but not by $Z_1(\gamma_j, A)$ as the example below shows.

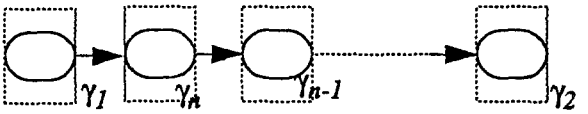

Fig. 1. Each bubble represents a set of states. Each highlighted rectangle represents a set of states which is passed to the recursive trim operator.

In the order tried γ_n, γ_{n-1}, ..., γ_1, no eliminations occur until γ_2, where γ_2 and γ_1 are eliminated. Repeated application eliminates all γ_j's. Obviously the order in which the γ_j's are tried is important for efficiency. Also, one can see that the elimination of γ_j's depends on the active set A. As A gets smaller, it becomes more likely that γ_j is a source or sink cycle set.

Remark There are several remarks about the recursive trim operator.

1) In general, the recursive trim operator would not return the same set, if the trim operator deleted only the source cycle sets or only the sink cycle sets.

2) There are situations in which deleting only sinks or sources suffice. For example, consider the following figure. If trim only deleted sources or sinks, all vertices are eliminated. However, applying both tests in trim can sometimes speed up the test. Assume, we are given the sets in the order $(\gamma_1, \ldots, \gamma_n)$. If trim deleted sinks, the computation takes $O(n^2)$ time, whereas if trim deleted both sinks and sources, the computation would take $O(2n)$. Hence, applying both tests in trim can sometimes speed up the test.

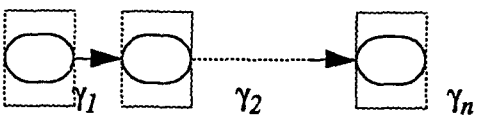

Fig. 2. Each bubble represents a set of states. Each highlighted rectangle represents a set of states which is passed to the recursive trim operator.

3) In our implementation, before calling recursive trim, we sort the cycle sets by the number of states they contain. We start processing the cycle sets from big to small, with the hope that bigger cycle sets will have a better chance of being a source or a sink.

2.3.2 The EL Algorithm

In this section, we will describe the adaptation of the Emerson-Lei method for the language containment check as it first appeared in [Tou91]. We will denote this algorithm by ***EL***. This algorithm calculates the set NC^+ as defined below.

Definition Let NC^+ (NC stands for not contained) denote the set of states x such that there is a path from x to a bad cycle.

Note that the language containment check passes iff $NC^+ = \emptyset$. All the algorithms presented in this section work on Q, the product of the system and the property with the recur edges removed. The EL method calculates the set $E(x)$ by the fixpoint computation $E(x) = \nu(X, V).(\prod_j R_1(x, R^*(x, \overline{\gamma}_j \wedge X)))$ (the language containment check passes iff $E(x) = \emptyset$). Note that $R^*(x, \overline{\gamma}_j \wedge E)$ is the set of all vertices which start a path in E whose end-point is in $\overline{\gamma}_j \wedge E$. We will show that $E(x) = NC^+(x)$. Note that this demonstrates the correctness of the algorithm.

Definition Let x be a vertex of G. Define $SCC(x)$ to be the SCC which contains x. Similarly, if $x \in S$, where S is some CSCC of G, let $CSCC(x) = S$.

Definition Define by SC_G the graph where every SCC of G is replaced by a node. Note that SC_G is acyclic.

Definition Let $x \in V$. Let H be the set of vertices y reachable from x, such that $y \in S$, where S is some ASCC of G. Define by $AS_G(x)$ the graph where G is restricted to H. Again note that $AS_G(x)$ is acyclic.

Lemma Let $H = R_1(x, R^*(x, \overline{\gamma}_j \wedge V))$ for some graph G with vertices V, and some $c \subseteq V$. If $x \in H$, then $SCC(x) \subseteq H$.

Theorem $E(x) = NC^+(x)$.

2.3.3 The EL1 Algorithm

In this section, we describe our first modification of EL. The EL1 method calculates the set $E_1(x)$ by the fixpoint computation $E_1(x) = \nu(X, V).(R_1(x, \prod_j R^*(x, \overline{\gamma}_j \wedge X)))$.

Theorem $E_1(x) = NC^+(x)$.

2.3.4 The EL2 Algorithm

Although one can present EL2 as a nested fixpoint computation, we choose to present the algorithm in a more sequential manner for two reasons. The first reason is that we would like to think of the operators as deleting irrelevant portions of a graph. Hence, they work like hyper-planes, constraining our current set of possible bad states. The sequential presentation makes this point more clear. The second reason is that such nested computations involve rather long formulas, and may be hard to read (for example, our final algorithm would involve four nested computations).

Definition Define the ***forward bad-path operator*** by $F(x) = \prod_j R^*(x, \overline{\gamma}_j \wedge A)$, where A is the current active set. Note that if a state can reach a bad cycle, it is not deleted by this operator. So, sink cycle sets are deleted by this operator. Similarly, define the ***backward bad-path operator*** by $B(x) = \prod_j R^*(\overline{\gamma}_j \wedge V, y)$. Note, if a state is reached by a bad cycle, it is not deleted by this operator. So, source cycle sets are deleted by this operator.

EL2 computes the set $E_2(x)$ as follows.

1) Let $E_2(x) = R(x)$, i.e. the set of all reachable states.

2) Repeat until convergence is achieved

2.1) Apply forward bad-path operator, $E_2(x) = \prod_j R^*(x, \overline{\gamma}_j \wedge E_2)$.

2.2) Apply forward stable operator, $E_2(x) = S^*(x, E_2)$.

Theorem $E_2(x) = NC^+(x)$.

Proof Let $H = E_2 \cap \overline{NC^+}$. We will first show $H \subseteq NC^+$, i.e. $H = \emptyset$. Let S be a leaf of SC_H. We will show S is a leaf of SCC_{E_2}. If not, S can reach some vertices in NC^+. Hence, $S \subseteq NC^+$, which is in contradiction to the assumption that S is a subset of H. If S is a ASCC of E_2, then S is deleted by the forward stable operator. This is a contradiction to EL2 having converged. If S is some CSSC of E_2, then it is contained in some γ_j. Since S is a sink cycle set, S is deleted by the forward bad path operator. Again, this is in contradiction to EL2 having converged. We conclude that $H = \emptyset$.

Conversely, assume that $x \in NC^+(x)$. We need to show $x \in E_2(x)$. It suffices to show that if $NC^+(x) \subseteq Y$, then no vertices in Y is deleted by either operator. This is easy to see since every state in Y has a successor and hence is not deleted by the forward stable operator. Moreover, every state of Y can reach a bad cycle, and hence is not deleted by the forward bad-path operator (QED Theorem).

Definition Let NC^- denote the set of states x such that there is a path from a state y, involved in some bad cycle, to x.

Corollary If we replace the forward operators in EL2 with backward operators, EL2 computes NC^-.

2.4) Early Cycle Detection

In practice, it is expected that the algorithm will be applied frequently with properties which fail. Hence, we would like to have special checks to find easily detectable bad cycles early. Let $\Gamma = \bigcup_j \gamma_j$. We classify the cycles of G into three groups:

1) Cycles which lie entirely in $\overline{\Gamma}$, i.e. ***cycles of the first kind***. Any cycle of the first kind is a bad cycle.

2) Cycle which intersect both Γ and $\overline{\Gamma}$, ***cycle of the second kind***. All such cycles are bad.

3) Cycles which are completely contained in Γ, i.e. ***general cycles or cycles of the third kind***. These cycles may be bad or good.

Definition Let $\Gamma = \bigcup_j \gamma_j$. Denote by $FC(x)$ (for first kind) the set of all state in $\overline{\Gamma}$, which can reach some cycle which lies entirely in $\overline{\Gamma}$, i.e. a cycle of the first kind. Let $CS_B(x)$ (for second kind) be the set of all states in $\overline{\Gamma}$, which are involved in some cycle of the second kind, whose length is less than B, where $B \geq 1$. Note that $CS_1(x) = FC(x)$. Similarly, let $CS(x) = \bigcup_n CS_n(x)$ be the set of all states in $\overline{\Gamma}$, which are involved in some cycle of the second kind.

Finding Cycles of the First Kind

1) **Let** $\Gamma = \bigcup_j \gamma_j$, **where each** γ_j **is a cycle set.**

2) **Let** $\hat{T}(x,y) = \overline{\Gamma(x)} \wedge T(x,y) \wedge \overline{\Gamma(y)}$, **i.e. the transition function restricted to** $\overline{\Gamma}$.

3) **Let** $F(x) = S^*(x, \overline{\Gamma})$.

Lemma Let $F(x)$ be the set returned by the above algorithm. Then, $F(x) = FC(x)$.

Remark In section 4, we pose a problem for which we don't have an efficient solution. The problem is finding the set of all states in a graph G which are in some CSCC. In other words, the set of all states which are involved in some cycle. Let this set be $C_G(x)$. If there are no cycles of the first kind, we have $CS(x) = \overline{\Gamma(x)} \wedge C_G(x)$.

We present the a method for finding cycles of the second kind after we introduce our final algorithm.

2.5 The Final Algorithm

The final algorithm first checks for cycles of the first kind, and then for short cycles of the second kind. If none is found, then it enters its main computation. First, recursive trim is called to reduce the state space as much as possible. Then, the main loop which consists of a forward and a backward pass is executed. The algorithm computes a set which is no larger than $NC^+(x)$.

Definition Let NC^{+-} denote the set of states x such that there are states y and z, involved in bad cycles, and x can reach y and z can reach x.

Final Algorithm

1) **Check for cycles of the first kind. If found, call the debugger.**
2) **Check for cycles of the second kind. If found, call the debugger.**
3) **Let** $F^*(x) = R(x)$, **where** $R(x)$ **is the set of reachable states.**
4) **Apply recursive trim operator,** $F^*(x) = Z^*(x,R) = \nu(Y,R).Z_1(\gamma_1, Z_1(\gamma_2, \ldots Z_1(\gamma_n, Y)\ldots))$.
5) **Repeat until convergence**
   - 2.1) **Apply forward bad-path operator,** $F^*(x) = \prod_j R^*(x, \overline{\gamma}_j \wedge F)$.
   - 2.2) **Apply forward stable operator,** $F^*(x) = S^*(x, F^*)$.
   - 2.1) **Apply backward bad-path operator,** $F^*(x) = \prod_j R^*(\overline{\gamma}_j \wedge F^*, y)$.
   - 2.2) **Apply backward stable operator,** $F^*(x) = S^*(F^*, y)$.
6) **If** $F^*(x) \neq \varnothing$, **call the debugger.**

Theorem Let $F^*(x)$ be the set returned by the above algorithm. Then, $F^*(x) = NC^{+-}(x)$.

Remark It is possible that after applying one of the operators in the loop of step 5 of the final algorithm, recursive trim operator can delete some vertices. For example, consider the situation illustrated by the following figure, where each bubble represents a SCC.

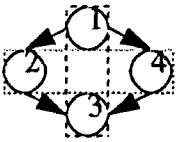

Fig. 3. Each bubble represents a SCC. Each highlighted rectangle represents a cycle set.

If recursive trim is applied to the above graph, no vertices would be deleted. However, if forward bad path operator is applied to this graph first, vertices in component 3 are removed. If we now apply recursive trim, all vertices are deleted. More experimentation is needed to justify whether recursive trim should be applied inside the loop of step 5.

2.6 Finding Cycles of The Second Kind

The algorithm for finding cycles of the second one returns the following set.

Definition Let CS^{+-} denote the set of states x such that there are states z and y in $CS(x)$, and there is a path from x to y and from z to x.

Finding Cycles of The Second Kind

1) **Check for cycles of the first kind. If none are found, go to step 2.**
2) **Apply step 4 of the final algorithm on G with the cycle set $\Gamma = \bigcup_j \gamma_j$.**

Remark If the above procedure is used to find cycles of the second kind, we can initialize F to $R \wedge \Gamma$. The reason is that since there are no cycles of the first or second kind, the states in $\bar{\Gamma}$ cannot be involved in any bad cycles.

3 Experimental Results

Up to now, we have performed experiments only on a limited set of examples. Four examples were based on the encyclopedia version of the dining philosophers problem, ranging from 16 to 40 philosophers. We checked for starvation, i.e. if a philosopher becomes hungry s/he is eventually fed. The number of reachable states for an n-philosopher example is about 2^n. Two examples were counters of size 100 and 500 states. In counters, we checked that the edge $(n,0)$ is taken infinitely often. The last example was part of an industrial design.

Seven different algorithms for language containment were compared: using iterated squaring by shifting in our first algorithm to compute transitive closure (iss), iterated squaring by folding in our first algorithm to compute transitive closure (isf), the original algorithm (org), and the algorithms described in section 2 EL, EL1, EL2, and final. The results show that final performs the best on our examples, except for counters where iterated squaring did very well. In general, iterated squaring by folding is faster than iterated squaring by shifting and linear computation, but is more space consuming. Iterated squaring by shifting first ran out of memory on 32 philosophers, where iterated squaring by folding ran out of memory at 48 philosophers. We also experimented with different properties on both the philosopher examples and the counters where the checks fail. On all these examples, the early failure detection algorithms found a bad cycle. Indeed, all bad cycles found were of the first kind. The following table summarizes our results.

	iss	isf	org	EL	EL1	EL2	final
phil16	118	55	114	9.8	6.8	6.7	3.5
phil24	535	290	442	25.9	20	20	9.9
phil32	Mem Out	670	1170	53	42	38	19
phil40	Mem Out	1625	2540	88	73	65	32
cnt100	1.4	1.6	9.7	2.9	2.2	2.2	2.2
cnt500	11.4	15.6	317	48	48	55	55
indus	8.1	7.2	8.5	3.0	4.4	3.5	.2 *

iss: iter sq by shifting in first alg
isf: iter sq by folding in first alg
org: first alg
EL: original Emerson-Lei
EL1: first modification
EL2: second modification
final: final alg

Time: reported in seconds

*: Check fails, and early cycle detection finds the error. Without early cycle detection, the algorithm takes 7.2 seconds.

Table 1

4 BDD Complexity Classes

As we have seen so far, developing efficient general purpose graph manipulation algorithms using BDD's has proven useful in solving our problems. To make the notion of efficiency more concrete when dealing with BDD's, we introduce the following definition. Possibly, this can serve as a guideline for designing efficient algorithms.

BDD Complexity Classes (BCC): Let finite domains $D_1, ..., D_n$ be given. Let ϕ be a formula describing a fixpoint computation involving predicates over the finite domains $D_1, ..., D_n$. Let the ***alternation depth*** of ϕ be as defined in [Eme86], which is roughly the number of alternations of μ and ν's. Then, the computation described by ϕ is in $BCC_{m,n}$ if the following holds:

1) The maximum arity of predicate variables in ϕ over which fixpoint computations are taken is less than or equal to m.

2) The alternation depth of ϕ is less than or equal to n.

Note that if there are no fixpoint computations in ϕ, then ϕ is in $BCC_{0,0}$. For example, taking the intersection or union of two sets of vertices is in $BCC_{0,0}$. Computing the set of reachable states is in $BCC_{1,1}$; and computing the transitive closure of a graph is in $BCC_{2,1}$. The EL algorithm of section 2.2.2. is in $BCC_{1,2}$. Based on our experience, it appears that the algorithms in $BCC_{m,j}$ are more time consuming than algorithms in $BCC_{n,j}$, where $m \geq n$. Also, in general, algorithms in $BCC_{j,m}$ are more time consuming than algorithms in $BCC_{j,n}$, where $m \geq n$. We pose the following problem:

Cycle Problem: Find a $BCC_{1,2}$ algorithm which finds the set of all vertices in a graph which are involved in some cycle.

Note that the above problem has an efficient (linear time) classical algorithm: find the SCC's and return all states except those in singleton SCC's with no self-loops. However, this algorithm is not efficient when BDD's are used. One application of an efficient solution to this problem is finding cycles of the second kind. Another application of this problem is in finding general cycles. Let $B(x)$ be the set returned by our final algorithm. Running the algorithm for the cycle problem would delete the

vertices in $B(x)$ which can reach and are reached by bad cycles but are not themselves involved in any cycle. This can potentially decrease the size of $B(x)$.

5 Conclusion

In this paper, we have presented several ways of speeding up the ω-regular language containment check using BDDs. By introducing five operators which trim the current active space, we are able to obtain very good result on our set of examples. The operators are forward bad-path, backward bad-path, forward stable, backward stable and trim. Special checks are also applied to find easily detectable failures early. On all of our examples, when the check failed, the bad cycles were found with these special checks. It is not clear what fraction of failures in practice will be caught by the early failure detection algorithms. We also introduced iterated squaring by folding as a method to speed up the transitive closure computation, which is needed in the algorithm described in [Tou91] for checking language containment. Finally, an open problem, i.e. a $BCC_{1,2}$ algorithm to find the set of states involved in some cycle in a graph was posed.

References

[Bou91] G. Boudel, V. Roy, R. de Simone, D. Vergamini, *"Process Calculi, from Theory to Practice: Verification Tools"*, in Automatic Verification Methods for Finite State Systems, Joe Sifakis ed., LNCS 407, 1989.

[Bur90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *"Symbolic Model Checking: 10^{20} states and Beyond"*. Logic in Computer Science, 1990.

[Ch74] 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, 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.

[Eme86] E. A. Emerson, C. L. Lei. *"Efficient Model Checking in Fragments of the Prepositional Mu-Calculus"*, In Symp. on Logic in Computer Science. IEEE, June 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.

[Cla90] E. M. Clarke, I. A. Draghicescu, R. P. Kurshan, *"A Unified Approach for Showing Containment and Equivalence Between Various Types of ω-Automata"*. In Proceedings of Fifteenth Colloquium on Trees in Algebra and Programming, 1990.

[Har90] Z. Har'El, R. Kurshan. *"Software for Analytical Development of Communications Protocols"*, ATT technical journal, 1990.

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

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

[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.