A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles*

Kavita Ravi[1], Roderick Bloem[2], and Fabio Somenzi[2]

[1] Cadence Design Systems
`kravi@cadence.com`
[2] University of Colorado at Boulder
{`Roderick.Bloem,Fabio`}`@Colorado.EDU`

Abstract. Detection of fair cycles is an important task of many model checking algorithms. When the transition system is represented symbolically, the standard approach to fair cycle detection is the one of Emerson and Lei. In the last decade variants of this algorithm and an alternative method based on strongly connected component decomposition have been proposed. We present a taxonomy of these techniques and compare representatives of each major class on a collection of real-life examples. Our results indicate that the Emerson-Lei procedure is the fastest, but other algorithms tend to generate shorter counter-examples.

1 Introduction

Model checking of specifications written in temporal logics such as LTL and fair CTL, or specified as ω-automata involves finding *fair cycles*. Since most model checking tools use one of these specification formalisms [16, 14, 2], detecting fair cycles is at the heart of model checking.

The most popular type of ω-automaton is the (generalized) Büchi automaton [3]. The fairness condition of a Büchi automaton consists of several sets of accepting states. A run is accepted if contains at least one state in every accepting set infinitely often. Consequently, the language of the automaton is nonempty iff the automaton contains a *fair cycle*: a (reachable) cycle that contains at least one state from every accepting set, or, equivalently, a *fair strongly connected component*: a (reachable) nontrivial strongly connected component (SCC) that intersects each accepting set.

LTL model checking [15, 23], language containment based on L-automata [14], and CTL model checking with fairness constraints [6] all reduce to checking emptiness of the language of a Büchi automaton. Hence, the core procedure in many model checking algorithms is to find a fair SCC if one exists, or to report that no such SCC exists.

In typical applications, existence of an accepting run indicates failure of the property. In this case, it is essential that the user be given an accepting run as a *counter-example*, typically presented in the form of a finite *stem* followed by a finite cycle. The counter-example should be as brief as possible, to facilitate debugging. Finding the shortest counter-example, however, is NP-complete [10, 4].

* This work was supported in part by SRC contract 98-DJ-620 and NSF grant CCR-99-71195.

The traditional approach to determine the existence of a fair SCC is to use Tarjan's algorithm [19]. This algorithm is based on depth-first search and runs in linear time in the size of the graph. In order to do the depth-first search, it manipulates the states of the graph *explicitly*. Unfortunately, as the number of states variables grows, an algorithm that considers every state individually quickly becomes infeasible.

Symbolic algorithms [16] manipulate sets of states via their characteristic functions. They derive their efficiency from the fact that in many cases of interest large sets can be described compactly by their characteristic functions. In contrast to explicit algorithms, an advantage of symbolic algorithms, which typically rely on breadth-first search, is that the difficulty of a search is not tightly related to the size of the state space, but is more closely related to the diameter of the graph and the size of the symbolic representation.

Several symbolic algorithms have been proposed that use breadth-first search and compute a set of states that contains all the fair SCCs, without enumerating them [7, 21, 11, 22, 13]. We refer to such sets as *hulls* of the SCCs. A *SCC-hull* algorithm is one that returns an empty set when there are no fair SCCs and computes a SCC-hull otherwise.

Recently, a symbolic approach for SCC enumeration has been proposed [24]. This algorithm explicitly enumerates SCCs, but not states. The technique is based on the observation that the SCC containing a state v is the set of states with both a path to v and a path from v. The earliest application of this observation to symbolic algorithms known to these authors is in the witness generation procedure of [10]. Xie and Beerel have proposed its use as the primary mechanism to identify SCCs. A similar algorithm, with a tighter bound on complexity, is proposed in [1]. We refer to these algorithms as *symbolic SCC-enumeration algorithms*.

Cycle-detection algorithms have mostly been presented in isolation, with their own proofs of correctness and limited comparison to other algorithms. In Section 3, we shall provide a comprehensive framework for SCC-hull algorithms, which is used to characterize the possibilities for cycle-detection algorithms, and to show correctness for any algorithm in this class. We discuss the relative merits of the currently known cycle-detection algorithms in Section 4 and evaluate their efficiency on a set of practical examples in Section 5. We report two important measures: The time it takes to decide whether a fair cycle exists, and the length of the counter-example that is offered to the user for debugging. We conclude the paper with Section 6.

2 Preliminaries

A graph is a pair $G = (V, E)$, with V a finite set of *states*, and $E \subseteq V \times V$ the set of *edges*. A *path* from $v_1 \in V$ to $v_k \in V$ is a sequence $(v_1, \ldots, v_k) \in V^+$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. If a path from u to v exists, we say that u *reaches* v. The *distance* from u to v is the length of a shortest path from u to v if u reaches v; infinity otherwise. The *diameter* d of G is the largest finite distance between two states in V, and n is the number of states.

A *strongly connected component* (SCC) of G is a maximal set of states $U \subseteq V$ such that for each pair $u, v \in U$, u reaches v. An SCC U is *trivial* if the subgraph of G induced by U has no edges. A set $V' \subseteq V$ is *SCC-closed* if every SCC that intersects V' is wholly contained in it.

The SCCs of G (both trivial and non-trivial) partition V into equivalence classes. Let $[v]$ denote the SCC U such that $v \in U$. The *SCC quotient graph* is a graph (W, H), such that W is the set of the SCCs of G, and $(U_1, U_2) \in H$ if and only if $U_1 \neq U_2$ and there exist $u \in U_1$, $v \in U_2$ such that $(u, v) \in E$. We define h to be the length of the longest path in the SCC quotient graph.

The quotient graph is directed and acyclic; hence it defines a partial order on the SCCs such that $[u] \leq [v]$ if and only if u reaches v. The sources of the graph are the minimal or *initial* SCCs, while the sinks of the graph are the maximal or *terminal* SCCs. The length of the longest reachable path through the SCC quotient graph is denoted by h. The number of reachable SCCs in a graph is denoted by N and the number of reachable non-trivial SCCs by N'. Note that $h \leq N$.

Let $C = \{c_1, \ldots, c_m\} \subseteq 2^V$ be a set of *Büchi fairness conditions*. An SCC U is *fair* if it is not trivial and $U \cap c_i \neq \emptyset$ for $1 \leq i \leq m$.

Given some encoding of the set of states, *symbolic graph algorithms* operate on the characteristic functions of sets of states and sets of edges. They achieve efficiency by manipulating many states at once. In particular, they can compute all the successors and predecessors of a set of states in a graph in time and space that depends more on the sizes of the characteristic functions than on the number of states. Each such computation is called a *step*. Computing all successors or predecessors in a single operation naturally leads to the use of breadth-first search (BFS) in symbolic algorithms. Depth-first search, by contrast, is not well-suited to symbolic implementation.

Throughout the paper, complexity bounds will be given in steps. Relating the maximum number of steps to the number of states n is not very indicative of performance for practical model checking experiments, for which n can be very large. Hence, we shall concentrate on bounds restricted to the quantities d, h, $|C|$, N, N', and $|W|$.

2.1 The Propositional μ-Calculus

Symbolic computations are conveniently described by formulae of the μ-calculus [5, 17], which is obtained from first-order predicate logic by adding the least (μ) and greatest (ν) fixpoint operators. Let $\mathcal{V}$ be a set of variables and A a set of atomic propositions. We use the following syntax, which corresponds to the propositional μ-calculus.

– If $f \in A \cup \mathcal{V}$, then f is a formula;
– If $f \in A$, then $\neg f$ is a formula;
– if f and g are formulae, so are $f \wedge g$, $f \vee g$, $\mathsf{EX}\, f$, $\mathsf{EY}\, f$, $\mathsf{AX}\, f$, and $\mathsf{AY}\, f$;
– if $Z \in \mathcal{V}$ and f is a formula, then $\mu Z.f$ and $\nu Z.f$ are formulae.

The semantics of μ-calculus are defined with respect to a labeled graph (G, L), where $G = (V, E)$ is a graph and $L : V \to 2^A$ is a labeling function.

Let $\mathcal{E} = \{e : \mathcal{V} \to 2^V\}$ be the set of *environments*, that is, the set of functions that associate a set of states to each variable, and let Φ be the set of μ-calculus formulae over A and $\mathcal{V}$. Then function $\mathcal{I} : \Phi \times \mathcal{E} \to 2^V$ associates a set of states to a formula $\varphi \in \Phi$ in a given environment $e \in \mathcal{E}$. Let $e[V'/Z]$ be the environment that coincides with e, except that $e[V'/Z](Z) = V'$. Function $\mathcal{I}$ extends $\mathcal{E}$ in the natural way, where

$$\begin{aligned}
\mathcal{I}(\mathsf{EX}\,f, e) &= \{v \in V : \exists (v, v') \in E, v' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(\mathsf{AX}\,f, e) &= \{v \in V : \forall (v, v') \in E, v' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(\mathsf{EY}\,f, e) &= \{v \in V : \exists (v', v) \in E, v' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(\mathsf{AY}\,f, e) &= \{v \in V : \forall (v', v) \in E, v' \in \mathcal{I}(f, e)\},\\
\mathcal{I}(\mu Z.f, e) &= \bigcap\{V' \subseteq V : V' \supseteq \mathcal{I}(f, e[V'/Z])\}, \text{ and}\\
\mathcal{I}(\nu Z.f, e) &= \bigcup\{V' \subseteq V : V' \subseteq \mathcal{I}(f, e[V'/Z])\} \quad .
\end{aligned}$$

We use the following abbreviations, which correspond to the familiar future and past tense CTL operators:

$$\begin{aligned}
\mathsf{EG}\,f &= \nu Z.\, f \wedge \mathsf{EX}\,Z & \mathsf{EH}\,f &= \nu Z.\, f \wedge \mathsf{EY}\,Z\\
\mathsf{E}\,f\;\mathsf{U}\;g &= \mu Z.\, g \vee (f \wedge \mathsf{EX}\,Z) & \mathsf{E}\,f\;\mathsf{S}\;g &= \mu Z.\, g \vee (f \wedge \mathsf{EY}\,Z)\\
\mathsf{EF}\,f &= \mathsf{E}\,\mathbf{true}\;\mathsf{U}\;f & \mathsf{EP}\,f &= \mathsf{E}\,\mathbf{true}\;\mathsf{S}\;f \quad .
\end{aligned}$$

The EX (or *preimage*) operator maps a set of states to the set of their direct predecessors. This corresponds to one step of backward symbolic BFS. Similarly, EY (*image* or a forward step) maps a set of states to the set of all their direct successors. The operators EX, AX, and all the abbreviations defined in terms of them are called *future-tense* or *backward* operators. EY, AY, and all the abbreviations defined in terms of them are called *past-tense* or *forward* operators.

3 Computing the Fixpoints

The evaluation of μ-calculus formulae requires the computation of fixpoints. Of particular interest to us are the following formulae.

$$\nu Z.\ \mathsf{EX} \bigwedge\nolimits_{c \in C} (\mathsf{E}\,Z\;\mathsf{U}\;(Z \wedge c)) \tag{1}$$

$$\nu Z.\ \mathsf{EY} \bigwedge\nolimits_{c \in C} (\mathsf{E}\,Z\;\mathsf{S}\;(Z \wedge c)) \quad . \tag{2}$$

These two equations describe SCC hulls. Specifically, (1) describes the set of states that can reach a fair cycle, and (2) describes the set of states that can be reached from a fair cycle. Both these sets include the states contained in a fair SCC. Moreover, the terminal SCCs included in (1) and the initial SCCs included in (2) are guaranteed to be fair. We generalize 1 and 2 to a family of μ-calculus formulae that can be used to find the SCCs of a graph. In Section 4, we will show that most known SCC-hull algorithms fall within this class.

The method of successive approximations [12, 20] can be used for the direct evaluation of fixpoints of continuous functions. It is often the case, however, that efficiency can be gained by transforming the given formula into an equivalent one for which, for instance, convergence of the approximations is faster.

3.1 Order of Evaluation in Fixpoints

Formulae (1) and (2) have the form

$$\nu Z.\, \tau_0\Big(\bigwedge_{1 \le i \le n} \tau_i(Z)\Big) \enspace . \tag{3}$$

Each τ_i is monotonic. (That is, $x \le y$ implies $\tau_i(x) \le \tau_i(y)$.) In addition, $\tau_1, \dots, \tau_n$ are downward:

Definition 1. *A function $\tau : X \to X$ is downward if, $\forall x \in X, \tau(x) \le x$.*

We can eliminate the conjunction from (3) thanks to the following result:

Theorem 1. *For $T = \{\tau_0, \dots, \tau_n\}$ a set of downward monotonic functions,*

$$\nu Z.\, \tau_0\Big(\bigwedge_{1 \le i \le n} \tau_i(Z)\Big) = \nu Z.\, \tau_0(\tau_1(\cdots(\tau_n(Z))\cdots)) \enspace .$$

Proof. (Sketch.) We prove by induction that

$$\bigwedge_{1 \le i \le n} \tau_i(Z) \ge \tau_1(\cdots(\tau_n(Z))\cdots) \enspace .$$

Letting $Z_\infty = \nu Z.\, \tau_0(\bigwedge_{1 \le i \le n} \tau_i(Z))$,we can then prove, again by induction, that, for $Z_k \ge Z_\infty$,

$$\tau_0(\tau_1(\cdots(\tau_n(Z_k))\cdots)) \ge Z_\infty \enspace .$$

The desired result follows easily. □

From [18] we know that for a set T of downward functions there is a unique fixpoint p such that every *fair* sequence over T has a finite prefix that converges to p. A sequence is fair if every element of T appears infinitely often in it. In our case, τ_i is downward for $i > 0$, but not for $i = 0$. However, if we change (1) as follows,

$$\nu Z.\, Z \wedge \mathsf{EX} \bigwedge_{c \in C} (\mathsf{E}\, Z \,\mathsf{U}\, (Z \wedge c)) \enspace .$$

and take $\tau_0 = \lambda Z.\, Z \wedge \mathsf{EX}\, Z$, then we have all downward functions, and we can apply them in any fair way. The transformation is guaranteed to preserve the fixpoint by Lemma 2.12 of [18]. The transformation can also be applied to (2).

The conjunction with Z that is used to turn τ_0 into a downward function is not always necessary, as shown by the following definition and lemma. (Cf. [13].)

Definition 2. *A set of states Z is* backward-closed *if $\mathsf{EF}\, Z = Z$; Z is* forward-closed *if $\mathsf{EP}\, Z = Z$.*

Lemma 1. *If Z is backward-closed, then $\mathsf{EX}\, Z \le Z$ and $\mathsf{EX}\, Z$ is backward-closed. If Z is forward-closed, then $\mathsf{EY}\, Z \le Z$ and $\mathsf{EY}\, Z$ is forward-closed.*

Therefore, if backward (forward) closure is a precondition for a computation of $\mathsf{EX}\,Z$ ($\mathsf{EY}\,Z$), then the conjunction of the result with Z is redundant.

It is always possible to make a given Z closed by removing edges from the graph. (This is done in [13] to deal with compassion constraints.) When forward (backward) closure is desired, the edges into (out of) states not in Z are removed. However, this is not necessarily better than making operators syntactically downward by adding a conjunction with Z.

3.2 A Generic Algorithm for Symbolic SCC-Hull Computation

We can combine (1) and (2) to compute a tighter SCC hull that contains only states that have a path both to and from a fair SCC.

$$\nu Z.\ \mathsf{EX}\,\mathsf{EY}\left[\bigwedge_{c \in C} (\mathsf{E}\,Z\,\mathsf{U}\,(Z \wedge c)) \wedge \bigwedge_{c \in C} (\mathsf{E}\,Z\,\mathsf{S}\,(Z \wedge c))\right]\ , \tag{4}$$

Formula 4 has the form of of (3). Hence, it is possible to apply the operators in any order, provided downwardness is guaranteed. This leads to the generic algorithm of Fig. 1.

```
GSH($G, I, T_F, T_B$) { /* graph, initial states, forward operators, backward operators */
    $Z := \mathsf{EP}\, I, \gamma := \emptyset$;
    do {
        $\zeta := Z$;
        $\tau :=$ FAIR_PICK($T_F - \gamma, T_B - \gamma$);
        $Z := \tau(Z)$;
    } until (CONVERGED($Z, \zeta, \tau, T_F, T_B, \gamma$))
    return $Z$;
}

CONVERGED($Z, \zeta, \tau, T_F, T_B, \gamma$) {
    if ($Z \neq \zeta$) {
        $\gamma := \emptyset$;
        return false;
    } else {
        $\gamma := \gamma \cup \{\tau\}$;
        return $T_F \subseteq \gamma \vee T_B \subseteq \gamma$;
    }
}
```

Fig. 1. Generic SCC-hull algorithm

In this algorithm, T_F is a set of downward forward operators that consists of $\lambda Z.\ Z \wedge \mathsf{EY}\,Z$ and $\lambda Z.\ \mathsf{E}\,Z\,\mathsf{S}\,(Z \wedge c)$ for each $c \in C$. Likewise, T_B is a set of downward backward operators that consists of $\lambda Z. Z \wedge \mathsf{EX}\,Z$, and $\lambda Z. \mathsf{E}\,Z\,\mathsf{U}\,(Z \wedge c)$ for each $c \in C$. Procedure FAIR_PICK guarantees fairness by ignoring operators in γ.

Procedure CONVERGED returns **true** if all operators of either T_F or T_B have been applied since the last time Z changed. Let P, Q, and R be the evaluations of (1) (the set of all states that can reach a fair SCC), (2) (the set of all states that can be reached from a fair SCC), and (4) (the set of states that can both reach and be reached from a fair SCC), respectively. Let $\hat{Z}$ be the evaluation of Z at convergence. Then,

$$R \leq \hat{Z} \leq P \vee R \leq \hat{Z} \leq Q \ .$$

This implies that $\hat{Z}$ is a hull of the fair SCCs of the graph. The condition that caused convergence (either $T_F \subseteq \gamma$ or $T_B \subseteq \gamma$) determines whether the initial SCCs or the terminal SCCs in $\hat{Z}$ are guaranteed to be fair.

Theorem 2. *GSH takes $O(|C|dN)$ steps.*

The proof follows along the lines of the proof of Theorem 1 in [1]. The next section shows how several practical algorithms can be obtained by choosing a specific strategy for FAIR_PICK. Simple strategies lead to the improved complexity bound of $O(|C|dh)$ steps and simplifications in the convergence.

4 A Taxonomy of Algorithms for Fair Cycle Detection

We are interested in BDD-based fair cycle detection algorithms for their ability to deal with large state graphs (albeit non-uniformly). Our taxonomy of the known algorithms is based on four factors—the μ-calculus representation of the state set, the complexity in terms of number of steps, the relationship between the computed set of states and the SCC quotient graph, and the length of the counter-examples.

If there are fair SCCs, the SCC-hull algorithms may return extra states besides those in the fair SCCs, depending on the specific μ-calculus formulae employed in the computation. The counter-example procedure is tightly connected to the computed state set. When there are fair SCCs in the state graph, the location of the fair SCCs must be known in order to generate the counter-example. In the SCC-enumeration approach, this is obvious since there is only one SCC in question. In the case of the SCC-hull algorithms, the position of the fair SCCs can be characterized depending on the formula applied. Different counter-example generation algorithms can be applied accordingly; the length of the counter-example generated may vary. When using BDDs, the sets explored in breadth-first search, commonly referred to as *onion-rings*, can be used to generate short counter-examples by deriving the shortest paths between pairs of states.

4.1 SCC-Hull Algorithms

In this section, we present a detailed comparison of algorithms that compute a hull, that is, a set of states that contains all fair SCCs. This computation starts with the set of all states and refines it until a fixpoint is reached. The fair SCCs are not enumerated in these algorithms. All but the Transitive Closure algorithm are instances of the generic SCC-hull algorithm of Fig. 1.

Algorithm of Emerson-Lei: The Emerson-Lei algorithm [7] computes the set of all states with a path to a fair SCC by evaluating the nested fixpoint formula

$$\nu Z. \bigwedge_{c \in C} \mathsf{EX}(\mathsf{E}\, Z\, \mathsf{U}\, (Z \wedge c)) \ .$$

Each EU ensures the reachability by all states in the current iterate Z of each fair set c. Each application of EX removes some states that do not reach cycles within the current set. The Emerson-Lei algorithm is an instance of the general algorithm presented in Section 3.2 that uses only the backward operators. The schedule of the EX and EU operators in this computation is fixed and fair. The application of the EX and all the EUs in each outer iteration improves the complexity bound of this algorithm to $O(|C|dh)$ steps [1].

The backward operators (EX and EU) compute the set that can *reach* fair SCCs and result in all terminal SCCs being fair. The hull defines a node cut on the SCC quotient graph, passing through fair SCCs, such that no SCC greater than an SCC on the cut is fair. The hull itself is defined by the set of SCCs less than or equal to the SCCs on the cut which may include unfair SCCs (nodes to the left of $\{4, 7, 8\}$ in Fig.2).

A counter-example procedure for this algorithm was suggested in [4]. Starting at the initial state, a path is found to the closest fair state. The search continues from one fair state to another until the path passes through all fair sets. Then an attempt is made to complete the cycle from the last fair state to the first fair state. If this attempt fails, i.e., if the first and last fair states belong to different SCCs, the procedure is restarted with the last fair state. A counter-example is guaranteed to be found since all terminal SCCs are fair. This algorithm attempts to find a cycle in a fair SCC close to the initial state. However there is no guarantee regarding the proximity to the initial states since terminal SCCs may be far from the initial states. This may lead to very long stems. The authors of [4] are aware of this drawback and leave the evaluation of the trade-off between finding shorter counter-examples and computational expense as an open issue.

Algorithm of Hojati/Kesten: Hojati et al. [11] (EL2 algorithm) and Kesten et al. [13] provide a fixed-schedule instance of the generalized algorithm of Fig. 1 that uses only forward operators. In our discussion we will refer to this algorithm as the EL2 algorithm. The fixpoint formula restricted to Büchi fairness is

$$\nu Z. \ \mathsf{EH} \bigwedge_{c \in C} (\mathsf{E}\, Z\, \mathsf{S}\, Z \wedge c) \ .$$

Each ES ensures reachability from a fair set within the current set. The ES operator is the temporal dual of the EU computation in the Emerson-Lei algorithm. The outer EH iteration removes states that are not reachable from a cycle within the set. While the forward dual of the Emerson-Lei algorithm has alternating EY and ES operations, the EH operation in the EL2 algorithm provides a different balance than the Emerson-Lei algorithm between reachability from cycles and reachability from fair sets. The different balance unfortunately disallows the $O(|C|dh)$ step complexity enjoyed by the Emerson-Lei algorithm. However, the EL2 method can be shown to perform better than the Emerson-Lei algorithm on some graphs with many trivial SCCs that satisfy all fairness constraints.

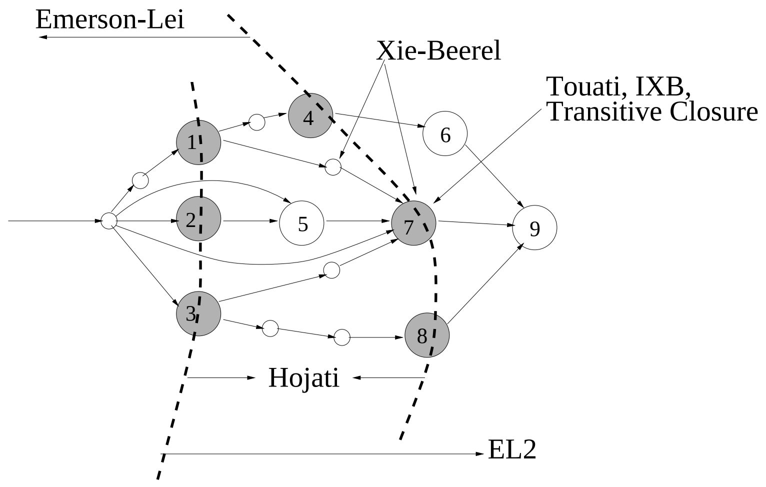

Fig. 2. An SCC quotient graph demonstrating resulting sets of the various methods. Shaded circles are fair SCCs. Small circles are trivial SCCs and large circles are non-trivial unfair SCCs.

The computed hull again defines a cut on the SCC-quotient graph. (Nodes to the right of $\{1, 2, 3\}$ in Fig. 2.) The cut passes through a set of fair SCCs such that there are no fair SCCs less than any SCC on the cut. The hull comprises the set of SCCs greater than or equal to those on the cut: The forward operators in the fixpoint algorithm compute a set that can be *reached from* the fair SCCs (the temporal dual of the hull computed by the Emerson-Lei algorithm) and result in all the initial SCCs being fair. The authors of [11] prove that if the operators were EG instead of EH and EU instead of ES, the resulting hull will be the same as in the Emerson-Lei algorithm. This observation is in agreement with Theorem 1.

A counter-example generation procedure for this hull is presented in [13]. The procedure first isolates one initial (fair) SCC in the hull, and then employs the counter-example procedure of [4] to find a path from the initial state to the isolated SCC. The isolation of the fair SCC to determine its location is necessary to apply the counter-example procedure of [4]. A similar approach is used in [10], with the difference that the isolation of the SCC is driven by the distance from the initial states rather than by the position in the quotient graph. It should be noted that an initial fair SCC is not necessarily closer to the initial states than a fair SCC that is not initial. (Node 7 in Fig. 2.) In the algorithm of [13] the choice of an initial SCC is mainly dictated by the requirement to deal with compassion constraints (Streett acceptance conditions).

Alternatively, the procedure of [4] can be applied to the hull produced by these algorithms by reversing the direction of edges in the state graph (causing the initial SCCs to become terminal) and picking any state in the set as an initial state. Once the cycle is identified, the stem from this cycle can be traced back to the initial states.

Algorithm of Hojati/Hardin: One of the algorithms presented in [11] and the one in [9] are close variants of each other. We discuss the procedure of [9] and refer to it as the

Hojati algorithm. Let

$$\mathsf{EB}\, f = \nu Z.\, f \wedge (Z \wedge \mathsf{EY}\, Z) \wedge \mathsf{EX}(Z \wedge \mathsf{EY}\, Z) \ \ .$$

The fixpoint representation of this algorithm is

$$\nu Z.\ \mathsf{EB}(\bigwedge_{c \in C} Z \wedge \mathsf{EF}(c \wedge Z) \wedge \mathsf{EP}(c \wedge Z)) \ \ .$$

The inner fixpoints ensure reachability of this set *from* and *to* the fair states, while the EB removes states that do not reach or cannot be reached from a cycle within this set. Unlike the previous two methods described here, this computation uses both the forward and backward operators. This procedure is also a fixed-schedule instance of the generalized algorithm of Fig. 1. The schedule is closer to the EL2 algorithm, being more balanced than the Emerson-Lei algorithm in terms of computing reachability to/from cycles and reachability to/from fair sets and consequently cannot have the $O(|C|dh)$ step complexity bound. The algorithm has a stricter convergence criterion than the generalized algorithm. This algorithm is purported by the authors to work best when there are no fair SCCs by converging to the empty set rapidly. The algorithms in [11] and in [9] vary slightly in the schedule in which the operators are applied.

The computed hull induces a subgraph of the SCC quotient graph, which is the intersection of the hulls induced by the Emerson-Lei algorithm and the EL2 algorithm. (Nodes to the right of $\{1, 2, 3\}$ and to the left of $\{4, 7, 8\}$ in Fig. 2.)

COSPAN uses the procedure of [4] to generate counter-examples for the Hojati algorithm. The counter-example procedure presented in [13] may be applied as well.

Transitive Closure Algorithm: Touati et al. developed this symbolic algorithm [21, 22] for the language emptiness check. This algorithm finds the fair SCCs by computing the transitive closure of the transition relation using iterative squaring. (Such a fixpoint computation is not expressible in propositional μ-calculus. It requires $O(\log d)$ transition relation expansions.) This identifies all states that lie on a fair cycle. The resulting set is exactly the set of fair SCCs and is a tighter set than any of the above methods. However, this method is not very efficient when using BDDs and can only be applied to small designs.

Although the fair SCC computation method is inefficient, the counter-example generation procedure proposed for this method is attractive. Touati et al. [21] propose the identification of a fair state (a state contained in some fairness constraint) on an SCC closest to the initial state. To do this they store the set of states reachable from the initial states and the corresponding onion rings. Then they generate the shortest stem to that fair state and complete the cycle through that fair state. The guarantee of the shortest prefix to a fair state on a cycle is possible because the computed fair set is exactly the fair SCCs. For example, this method may first identify 2, 3 or 7 as the fair SCCs in Fig.2 if one of these SCCs contains a fair state close to the initial state. Consequently, the closest fair state is guaranteed to lie on a fair cycle, although this may not be the fair SCC closest to the initial state. To guarantee a counter-example with a stem of minimal length, it is necessary to find a fair SCC closest to the initial states.

Summary: The SCC-hull algorithms compute sets of states that include all the fair SCCs using fixpoint computations. The Emerson-Lei algorithm has a better complexity bound ($O(|C|dh)$ steps) than the EL2 and Hardin methods ($O(|C|dN)$ steps) although the latter can be shown to perform better than the Emerson-Lei algorithm on some graphs. The hulls computed by the different algorithms and the location of the fair SCCs (important for counter-example generation) are determined by the operators applied in the fixpoint computations. The location of the fair SCCs along with the counter-example generation procedure determines the length of the counter-examples.

4.2 Symbolic SCC-Enumeration Algorithms

In this section, we present methods that enumerate the SCCs of a state graph but do not require explicit manipulation of the states. BDDs can be employed to enumerate the SCCs while taking advantage of the symbolic exploration of the states.
Algorithm of Xie-Beerel: Xie and Beerel [24] propose an algorithm that uses a BDD-based approach in enumerating the SCCs. The main observations that this algorithm uses are that

$$[v] = \mathsf{EF}(v) \wedge \mathsf{EP}(v) = \mathsf{EF}(v) \wedge \mathsf{E}\,(\mathsf{EF}(v)\;\mathsf{S}\;v)\enspace, \tag{5}$$

and that $\mathsf{EF}(v) \wedge \neg[v]$ is an SCC-closed set. The algorithm picks a random seed state v from the set of reachable states and computes $\mathsf{EF}(v)$ and $\mathsf{E}\,(\mathsf{EF}(v)\;\mathsf{S}\;v)$. The SCC $[v]$ is computed as defined in 5. If $[v]$ is found to be fair, the algorithm terminates. Otherwise, the remaining states are partitioned into $\mathsf{EF}(v) \wedge \neg[v]$ and $V \wedge \neg\,\mathsf{EF}(v)$. The algorithm first recurs on $\mathsf{EF}(v) \wedge \neg[v]$ and then on the $V \wedge \neg\,\mathsf{EF}(v)$, restricting the search to the current partition. A pruning procedure FMD_PRED is employed to prune away states in these two partitions that do not reach any non-trivial SCC. The pruning reduces the chances of picking a trivial SCC as the random seed. (The FMD_PRED computation is inefficient since it uses both an EX and an EY computation.) If there are no fair SCCs, all SCCs are eventually enumerated. The complexity of this algorithm is $O(Nd)$ steps. This algorithm can also be applied when the set of reachable states cannot be computed with complexity $O(|W|d)$ steps.

The random selection of a seed state does not provide a systematic enumeration order of the SCCs. For instance, the SCCs marked Xie-Beerel in Fig.2 may be the first ones identified in that graph. A counter-example may be generated by using the method of [4] to generate a prefix to the fair SCC and then finding a fair cycle within it. The length of the counter-example depends on which fair SCC is enumerated first, which cannot be anticipated with this algorithm.

The algorithm in [10] also uses the observation of 5 to compute SCCs. The onion rings for the reachable states are computed incrementally (this method can be applied when the reachable states are too large to compute) and a seed state is picked from the onion-rings starting from the initial states. This is similar to the idea in [21], except that the SCCs are computed as in 5 instead of using the transitive closure of the transition relation. However, the algorithm in [10] does not partition the state space during the search, resulting in $O(Nd)$ steps.
Improvements to the Xie-Beerel Algorithm: We propose two improvements over the Xie-Beerel algorithm that reduce its worst-case complexity and the length of the generated counter-example. The improved Xie-Beerel procedure (IXB) is shown in Fig. 3.

```
ENQUEUE ($Q$, $S$, $O$, $M$){ /* ENQUEUE (priority queue, set of states, onion-rings, mode) */
        if ($H \in M$) $S := \mathsf{EH}\, S$;
        if ($G \in M$) $S := \mathsf{EG}\, S$;
        if ($S$ does not intersect all fair sets) return ;
        $i$ := index of the innermost ring, $O_i$, that intersects $S$;
        insert $(S, i)$ in $Q$;   /* $S$ prioritized by $i$ */
}

IXB ($R$, $O$){ /* IXB( reachable states, onion rings )*/
        $Q$ := empty priority queue;
        ENQUEUE($Q, R, O, \{H, G\}$);
        while ($Q$ not empty) {
                $(V, i) :=$ pop($Q$);   /* contains a non-trivial SCC ($V \neq 0$) */
                $v$ := {random state picked in $V \cap O_i$};   /* may be a trivial SCC. */
                $B := \mathsf{E}\, V\, \mathsf{U}\, v$;
                $S := \mathsf{E}\, B\, \mathsf{S}\, v$;
                if ($S \neq 0$)
                        if ($S$ intersects all fair sets) return $S$;
                else $S := v$;
                ENQUEUE($Q, B \setminus S, O, \{G\}$);   /* does not need EH */
                ENQUEUE($Q, V \setminus B, O, \{H\}$);   /* does not need EG */
        }
}
```

Fig. 3. Pseudocode of the improved version of the Xie-Beerel algorithm. Parameter M to ENQUEUE defines the amount of reduction to be applied to S. Specifying $M = \{H, G\}$ is never wrong, but it may be inefficient. Algorithm IXB uses the onion rings of reachability analysis. It picks the seed in the innermost onion ring that intersects the state space

The two important enhancements are adding EH to the pruning of the partitions and examining the partitions prioritized by distance from initial state. The first improvement adds EH to the EG operator (corresponding to the FMD_PRED operator in Xie-Beerel). EH prunes states in the partitions that do not reach an SCC in addition to those that are not reachable from an SCC. The worst case complexity of this algorithm can be shown to improve from $O(Nd)$ to $O(N'd)$ due to this addition (Theorem 3 in [1]). The EG operator is more efficient than the FMD_PRED operator that the Xie-Beerel algorithm applies. Also the pruning is controlled by the argument M (mode). Since both modes of pruning (EG and EH) are applied to the initial set (reachable states) and every subsequent partition is either forward or backward closed, pruning only needs to be applied as EH (to forward-closed sets) or EG (to backward-closed sets).

The second improvement over Xie-Beerel is designed to produce shorter counter-examples. A priority queue Q is instantiated to maintain the partitions in the order of their distance from the initial states (computed using the onion-rings). Processing the partitions in the order of their distances from the initial state is guaranteed to yield a fair SCC closest to the initial state. This idea is the same as in [21]. In Fig.2, this method will first remove the initial trivial SCCs and its successor trivial SCC by trimming, and then identify one of 2, 3 and 7 as the fair SCC. The resulting SCC is also the same as the one produced by the algorithm in [10] if the picked seed states are the same, but this algorithm is more efficient since it maintains the partitions. While the Transitive closure procedure does not identify any unfair SCCs, this method may. The number of unfair SCCs examined may be reduced by picking a seed state that is close to the initial states and in many fairness conditions.

Both the original Xie-Beerel procedure and IXB first search backward from the seed state, and then use set B (Fig.3) to trim the forward search. A different algorithm is presented in [1] that uses a lockstep search. Here the forward and backward searches from the seed state are interleaved. The first that terminates is used to trim the other. Since the first that terminates may not be $\mathsf{EF}(v)$, IXB and the lockstep search may explore different partitions of the state space. (For more details, refer [1]).

Again, the procedure of [4] can be applied to generate a counter-example. Using the onion rings, the the shortest stem to any fair SCC can be computed. Since the precise location of a fair SCC is known, the complexity of generating the counter-example is reduced [1].

Summary: SCC-enumeration algorithms apply BDD-based approaches to enumerate the SCCs in the graph and apply pruning to reduce the number trivial SCCs examined. Unlike the SCC-hull approaches, fairness of the set being computed is checked after each SCC is identified. Some optimizations have been applied to improve the likelihood of examining fair SCCs, such as discarding partitions that do not intersect one or more fairness constraints. At an additional computational cost, the partition may be refined to contain only states that reach all fairness constraints.

The best SCC-hull algorithm has a $O(|C|dh)$ step bound, whereas the best symbolic SCC-enumeration algorithm has a $O(N'd)$ step complexity bound. The SCC-hull approach may perform better if the diameter is small, whereas the SCC-enumeration approach relies on a small number of (unfair) SCCs to outperform the SCC-hull algorithms. Apart from the worst case complexity, the performance of the algorithms is heavily dependent on the size of the BDDs that arise during the computations, which means that trends are not easily discerned. Experimental results are presented in the following section.

If some model-checking runs are repeated as in a regression test, one may have a pretty good idea of what approach will work better. In particular, if there are no fair cycles, an SCC-hull algorithm may be preferable to SCC enumeration. The expected result may be used to bias the decision between the two approaches.

5 Experimental Results

We implemented five of the algorithms discussed in Section 4—Emerson-Lei, EL2, Hojati, IXB, and Lockstep—in Cospan [8]. The objective was to compare and study the performance of these algorithms in checking language emptiness. All methods are implemented using BDDs. We did not consider the transitive closure algorithm in the experimental evaluation since it is infeasible for large examples. The IXB algorithm is regarded as representative of the Xie-Beerel method. The five implemented methods capture the key ideas in the fair SCC computation algorithms discussed in this paper.

Table 1. Comparison of computation times and number of steps for Emerson-Lei, EL2, Hojati, IBX1 and Lockstep methods. In Column 3, N stands for no fair SCC, F stands for fair SCC found. Reachability times is in seconds

Circuit	State	N/	Reach	EL		EL2		Hojati		IXB		Lockstep	
	Vars	F	Time	Time	EX /	Time	EX /	Time	EX /	Time	EX /	Time	EX /
			(sec)	(sec)	EY	(sec)	EY	(sec)	EY	(sec)	EY[SCC]	(sec)	EY[SCC]
fischer	4	N	0.06	0.48	170	0.43	275	0.47	247	0.42	205[24]	0.42	208[21]
cycle7	8	N	0.01	0.04	61	0.05	45	0.04	50	1.28	558[83]	2.07	926[125]
Abp2	16	N	0.01	20.32	352	16.52	720	30.91	733	58	2506[693]	59	1689[465]
CI	62	N	1989	4.53	5	4.46	5	8.78	12	4.10	5[0]	4.11	5[0]
F1	70	N	258.6	68.59	4	50.34	4	73.82	12	50.81	4[0]	50.48	4[0]
IR	91	N	1448	2.69	5	2.71	5	56.08	12	3.70	5[0]	3.64	5[0]
RR	94	N	66.5	12.97	50	12.54	50	88.10	58	11.61	50[0]	12.25	50[0]
cmr	97	N	103	0.03	2	0.03	2	0.05	10	0.05	2[0]	0.06	2[0]
S111	191	N	259	0.82	4	0.89	5	4.70	13	1.28	6[0]	1.28	6[0]
S195	483	N	222.5	1.89	3	1.74	3	10.47	10	17.30	3[0]	17.44	3[0]
cycle	7	F	0.1	0.6	183	0.6	232	0.77	305	0.3	16[1]	0.4	16[1]
S142	31	F	10.3	4.5	4	13.1	1285	7.5	12	13.2	1285[1]	13.1	1285[1]
S179	32	F	33.1	0.7	19	0.9	44	1.9	26	0.9	42[1]	0.9	42[1]
S192	44	F	1149	2440	204	117.6	484	4671	462	224.7	172[1]	235	172[1]
BC	70	F	15.5	0.2	20	0.5	14	1	24	0.4	13[1]	0.4	13[1]
S118	80	F	119.5	7	3	340.4	35	4917	26	430.4	34[1]	429.5	34[1]
S124	80	F	117.1	7.1	3	338.1	35	4990	26	425	34[1]	429.8	34[1]
WV	92	F	62.9	288.4	6	4807	16	8958	14	5937	14[1]	5779	14[1]
CO	93	F	773.8	198.5	27	9125	174	96.4	38	8755	174[1]	7179	174[1]
S119	93	F	156.1	137.1	16	1144	157	7720	102	564	45[1]	516	45[1]
S120	93	F	160.8	38	9	643.6	128	4050	74	406.4	43[1]	376.1	43[1]
S106	124	F	185	68.9	35	51	174	399	91	75.5	55[1]	76	55[1]

The main metrics in this comparison are the time taken to compute the fair set, the number of steps and the length of the generated counter-example. It is also interesting to compare the performance of SCC-hull algorithms against the symbolic SCC-enumeration algorithms in the presence and absence of fair SCCs. Additionally, the factors that are likely to play an important role in BDD-based computations are the order of variables and the number of dynamic reorderings and the size of the BDDs when the reorderings are triggered, the frequency of garbage collection of BDD nodes, and the availability of previous computed results in the cache and the unique table of the BDD package.

Tables 1 and 2 present results for 22 examples. These are industrial-strength designs of varying sizes; the size reported for each design is for the part relevant to the property being checked. Property satisfaction is verified using a standard language emptiness check. The number of fairness constraints ($|C|$) in these examples varies from 0 to 14. All experiments were conducted on an SGI machine and 2GB memory with standard options and dynamic reordering. The top half of Table 1 reports examples that do not have a fair SCC and the bottom half reports examples that do. Table 2 reports counter-example related data only for the examples that contain a fair SCC.

Table 1 provides a comparison of the times to compute the fair SCCs by the different algorithms. The first four columns indicate the name of the circuit, the number of state variables, whether a fair SCC was present or not, and the time taken to compute the reachable states. The remaining columns provide the time taken to compute the fair SCCs and the number of steps for each algorithm.

The first 10 examples have no fair SCCs. The SCC-hull algorithms compute an empty resulting set. The Emerson-Lei algorithm and the EL2 algorithm have similar performance and are better than the other methods. Although the Hojati method claims to work better on examples that do not contain a fair SCC, these examples show performance worse (by small factors) than the Emerson-Lei and the EL2 algorithms in more than half the cases. The IXB and the Lockstep methods have similar performance, but worse than the Emerson-Lei and EL2 methods. In cycle7 and Abp2, enumeration of a large number of unfair SCCs contributes to slower run times compared to the Emerson-Lei algorithm. In S195, the time difference is due to the different BDD sizes arising in the different computations, and from the additional pruning step (EH) of the IXB and Lockstep methods. The numbers of steps roughly correlate with the computation times of the various algorithms. The computation times are too small to clearly distinguish the merits of applying forward vs. backward operators, the interleaving of EX, EY, EF, and EP operations or computing the reachability from/to a cycle vs. reachability from/to a fair set.

The bottom half of Table 1 reports data for examples with a fair cycle. Here the Emerson-Lei algorithm performs better than other algorithms on most cases. The EL2 method is an order of magnitude slower on 5 of the 12 examples. The longer runtimes compared to the runtimes of the Emerson-Lei algorithm may be attributed to the larger number of steps, resulting in the increased allocation of BDD nodes, dynamic reorderings, and garbage collections. This may be due to the different state sets arising in the forward and backward computations or the interleaving of reachability to a fair set and reachability to a cycle. In the case of S192, although the number of steps is more than twice the number of steps in the Emerson-Lei method, the BDD sizes are smaller and the number of dynamic reorderings and garbage collections are fewer, therefore requiring a much smaller computation time.

The Hojati method has the worst performance on all examples except S142 and CO. Surprisingly, the computation times are often significantly larger even when the number of steps are fewer than, say, the EL2 algorithm. The justification of these data is supported by increased BDD sizes during the computation, more dynamic reorderings and garbage collections of the unique table. Apart from the state sets encountered during the computation, the inefficiency of the Hojati method in the reuse of previously computed results compared to the Emerson-lei or the EL2 method may explain the increased computation times. The inefficiency is due the interleaving of the EX and EY results in the computed table, causing the eviction of many useful results.

The performance of IXB and the Lockstep methods are tightly connected. In both methods, the first SCC found is fair. The speed for these two methods differs significantly only in a few examples. The number of steps performed are also the same. Hence

Table 2. Comparison of counter example generation times and length for Emerson-Lei, EL2, Hojati, IBX1 and Lockstep methods

Circuit	State	EL		EL2		Hojati		IXB		Lockstep	
	Vars	Time	(Prefix, Loop)	Time	(Prefix, Loop)	Time	(Prefix, Loop)	Time	(Prefix, Loop)	Time	(Prefix, Loop)
cycle	7	0.01	(7,7)	0.01	(2,7)	0.01	(2,7)	0.01	(2,7)	0.01	(2,7)
S142	31	1.64	(7,512)	1.62	(7,512)	2.56	(7,512)	1.68	(7,512)	1.60	(7,512)
S179	32	0.10	(10,6)	0.09	(10,6)	0.14	(10,6)	0.10	(10,6)	0.12	(10,6)
S192	44	1.38	(48,32)	0.77	(20,32)	1.52	(28,40)	0.79	(20,32)	0.78	(20,32)
BC	70	0.20	(7,2)	0.19	(6,2)	0.29	(8,2)	0.15	(6,2)	0.18	(6,2)
S118	80	0.60	(7,8)	0.82	(7,10)	1.69	(7,10)	0.73	(7,10)	0.71	(7,10)
S124	80	0.60	(7,8)	0.90	(7,10)	1.71	(7,10)	0.74	(7,10)	0.66	(7,10)
WV	92	0.64	(3,4)	1.44	(3,4)	12.05	(3,2)	6.11	(3,2)	6.07	(3,2)
CO	93	2.93	(13,60)	3.22	(13,32)	3.63	(13,46)	3.32	(13,32)	3.35	(13,30)
S119	93	1.62	(7,20)	1.66	(7,8)	2.80	(8,18)	1.63	(7,18)	1.43	(7,16)
S120	93	1.28	(8,18)	1.34	(7,8)	2.46	(9,16)	1.78	(7,18)	1.58	(7,18)
S106	124	3.11	(51,4)	2.84	(45,6)	4.81	(55,2)	2.95	(45,4)	2.89	(45,4)

the Lockstep and the IXB methods are most likely performing the same EX and EY operations in finding the fair SCC, albeit in different order. Compared to the Emerson-Lei algorithm, these methods are generally slower.

In summary, the Emerson-Lei method is the fastest and also has a better complexity bound than the EL2 and Hardin methods. Since different sets are computed by the different algorithms, the performance varies with different examples. The table also indicates that the BDD-related factors play an important role in the computation times. When the run times do not correlate with the number of steps, the BDD statistics provide an explanation for the slowdown or speedup.

In Table 2, we present the computation time and the length of counter-examples for the cases in which a fair SCC was found. The time taken to generate a counter-example is small compared to fair SCC computation time. The counter-examples generated by the EL2, IXB, and Lockstep methods have shorter prefixes than those of the Emerson-Lei algorithm. This suggests that the SCC identified is initial as well as close to the initial state. The reduction in a long prefix such as that of S192 is definitely desirable although a cycle of minimal length cannot be guaranteed by any of the algorithms.

In summary, when the counter-example generated by the fastest algorithm, Emerson-Lei, has a long prefix, then one of EL2, IXB, or Lockstep methods can be applied to find a shorter prefix to the fair cycle. Some hit in computation time is expected for this benefit.

6 Conclusions

We have discussed various symbolic algorithms to check for existence of fair cycles. These algorithms fall in two major classes: the SCC-hull and the SCC-enumeration methods. We have presented a framework for the SCC-hull algorithms, that allows us to easily derive other symbolic algorithms for fair-cycle detection, such as a forward version of Emerson-Lei's algorithm or a particular schedule of the generalized algorithm. Moreover, it allows us to quickly infer correctness of new algorithms.

We have presented a taxonomy and comparison of symbolic algorithms, and we have discussed the merits of the different counter-example generation routines.

Finally, we have compared five of the algorithms using examples from practice. We have found that none of the algorithms that have been proposed as alternatives to the one of Emerson-Lei performs better on a significant number of cases. If, on the other hand, the length of the counter-examples for debugging is the primary concern, the EL2, IXB, and Lockstep algorithms often produce better results.

In future work, we want to separate the influence on efficiency due to four factors—forward versus backward or mixed computations, the computation schedule, the counter-example generation routine, the use of don't-cares and the influence of BDD-related issues.

We plan to study a hybrid routine for fair-cycle detection. The hybrid method may split the state space, as the SCC-enumeration routines do, when the BDDs are large. Splitting allows it to simplify both the state set and the BDDs representing the graph. When the BDDs are simple, the procedure may switch to Emerson-Lei.

Acknowledgements We would like to thank Bob Kurshan for pointing us to this problem and for very useful suggestions, Ron Hardin for his help with Cospan and Peter Beerel for providing us with an early version of his paper.

References

[1] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In these proceedings.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[3] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[4] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.

[5] J. W. de Bakker and D. Scott. A theory of programs. Unpublished notes, IBM Seminar, Vienna, 1969.

[6] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[7] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[8] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Eighth Conference on Computer Aided Verification (CAV '96)*, pages 423–427. Springer-Verlag, 1996. LNCS 1102.

[9] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 268–278. Springer-Verlag, Berlin, 1997. LNCS 1254.

[10] R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV '93)*, pages 41–58. Springer-Verlag, Berlin, 1993. LNCS 697.

[11] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.

[12] L. Kantorovitch. The method of successive approximations for functional equations. *Acta Mathematica*, 71:63–97, 1939.

[13] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98)*, pages 1–16, Berlin, 1998. Springer. LNCS 1443.

[14] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[15] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[17] D. M. R. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5, 1970.

[18] F. Somenzi. Symbolic state exploration. *Electronic Notes in Theoretical Computer Science*, 23, 1999. http://www.elsevier.nl/locate/entcs/volume23.html.

[19] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[20] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[21] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. In *1991 International Workshop on Formal Methods in VLSI Design*, Miami, FL, January 1991.

[22] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.

[23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[24] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of the International Conference on Computer-Aided Design*, pages 37–40, San Jose, CA, November 1999.