Sharp Disjunctive Decomposition for Language Emptiness Checking ⋆

Chao Wang and Gary D. Hachtel

Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO, 80309-0425
{wangc,hachtel}@Colorado.EDU

Abstract. We propose a "Sharp" disjunctive decomposition approach for language emptiness checking, which is specifically targeted at "Large" or "Difficult" problems. Based on the SCC (Strongly-Connected Component) quotient graph of the property automaton, our method partitions the entire state space so that each state subspace accepts a subset of the language, the union of which is exactly the language accepted by the original system. The decomposition is "sharp" in the sense that it allows BDD operations on the concrete model to be restricted to small subspaces, and that unfair and unreachable parts of the submodules and automaton can be pruned away. We also propose a "sharp" guided search algorithm for the traversal of the state subspaces, with its guidance the approximate distance to the fair SCCs. We give experimental data showing that our new algorithm outperforms previously published algorithms, especially for harder problems.

1 Introduction

Language emptiness checking on the fair Kripke structure is an essential problem in LTL [11, 15] and fair-CTL [12] model checking, and in the language-containment based verification [10]. Symbolic fair cycle detection algorithms – both the SCC-hull algorithms [6, 7, 14, 8] and the SCC enumeration algorithms [17, 1], can be used to solve this problem. However, checking language emptiness is harder than checking invariants, since the later is equivalent to reachability analysis and has a linear complexity. Due to the well-known *state space explosion*, checking language emptiness can be prohibitively more expensive and is still considered to be impractical on industry scale circuits.

Symbolic fair cycle detection requires in the general case a more than linear complexity: $O(n^2)$ for SCC-hull algorithms and $O(n \log n)$ for SCC enumeration algorithms, where n is the number of states. For those cases where the automata are *weak* or *terminal* [9, 2], special model checking algorithms usually outperform the general ones. This idea has been further extended by [16], which combines compositional SCC analysis with specific decision procedures tailored to the cases of strong, weak, or terminal automata. It thus takes advantage of those strong automata with weak or terminal SCCs, and of those strong SCCs that turn into weak or terminal SCCs after the automata are composed with the model.

⋆ This work was supported in part by SRC contract 2001-TJ-920 and NSF grant CCR-99-71195.

In the algorithm of [16], SCC analysis is also used during the localization reduction to limit BDD attention to one fair SCC of the partially composed abstract model at a time. This permitted BDD restriction to a small state subspace during expensive operations that needed to be performed on the entire concrete model.

By partitioning the sequential system into subsystems and inspecting each of these small pieces separately, the chance of solving the problem may increase. In the context of reachability analysis, Cho et al. [4] proposed the *machine decomposition* algorithm: it partitions the sequential system using its latch connectivity graph, so that each subsystem contains a subset of latches of the original system.

For language emptiness checking, we propose in this paper a new algorithm for state space decomposition that is based on the notion of *sharpness*. Our algorithm partitions the original state space S into a collection of state subspaces S_i, according to the SCC quotient graph structure of the *amassed* property automaton. A nice feature of these state subspaces is that, each of them can be viewed as a separate fair Kripke structure. Furthermore, if we use $L(S)$ to represent the original language, and $L(S_i)$ to represent the language accepted within each state subspace, we have $L(S_i) \subseteq L(S)$ and $\cup_i L(S_i) = L(S)$. This allows us to check language emptiness on each state subspace separately. Our decomposition is "sharp" in that the BDD operations on the concrete model are focused on very small state subspaces, and in the sense that unfair and unreachable parts of the submodules and automaton can be pruned away.

We further propose a "sharp" forward (and backward) guided search algorithm for the traversal of the sate subspaces, which uses the approximate distance to the fair SCCs to guide the search. At each breadth-first search step, we only compute a subset of normal image with a smaller BDD size (sharp) and a closer distance to the potential fair SCC (guided). Whenever the reachable subset intersects a *promising* state – a state that is in the fair SCC-closed set and satisfies some fairness constraints – we use that state as a seed to enumerate the SCC. We stop as soon as an accepting (or fair) SCC is found, knowing that the language is not empty. If we cannot find any fair SCC before the forward search reaches a fix-point, or all th fair SCC-closed sets has been explored, we know the language is empty. Note that our new algorithm does not use the weak/terminal automata strength reduction techniques of [2].

On practical circuits, reachability analysis or even a single image computation can be prohibitively expensive. In fact, our new algorithm is directed specifically toward such larger problems. Thus, it is to be expected that algorithms with less heuristic overhead might outperform our "sharp" algorithm for easily soluble problems. The experimental results show this, but they also show that when the language emptiness problems become harder, our "sharp" algorithm outperforms both Emerson-Lei (the standard language emptiness checking algorithm in VIS [3]) and D'n'C [16].

The flow of the paper is as follows. We present the basic definitions in Section 2, followed by the state space decomposition theory in Section 3. In Section 4, we describe our new algorithm and analyze its complexity. The experimental results are given in Section 5, and we conclude and discuss potentially fruitful future work in Section 6.

2 Preliminaries

We combine the model $\mathcal{M}$ and the property automaton $\mathcal{A}_{\neg\psi}$ together, and represent the entire system as a *labeled, generalized* Büchi automaton [1] $\mathcal{A} = \mathcal{M} * \mathcal{A}_{\neg\psi}$.

Definition 1. *A* labeled, generalized *Büchi automaton is a six-tuple*

$$\mathcal{A} = \langle S, S_0, T, \mathcal{F}, A, \mathcal{L} \rangle \ ,$$

where S is the finite set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, $\mathcal{F} \subseteq 2^S$ is the set of fairness conditions, A is the finite set of atomic propositions, and $\mathcal{L} : S \rightarrow 2^A$ is the labeling function.

A *run* of $\mathcal{A}$ is an infinite sequence $\rho = \rho_0, \rho_1, \ldots$ over S, such that $\rho_0 \in S_0$, and for all $i \geq 0$, $(\rho_i, \rho_{i+1}) \in T$. A run ρ is *accepting* if, for each $F_i \in \mathcal{F}$, there exists $s_j \in F_i$ that appears infinitely often in ρ.

The automaton accepts an infinite word $\sigma = \sigma_0, \sigma_1, \ldots$ in A^ω if there exists an accepting run ρ such that, for all $i \geq 0$, $\sigma_i \in \mathcal{L}(\rho_i)$. The language of $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the subset of A^ω accepted by $\mathcal{A}$. The language of $\mathcal{A}$ is nonempty iff $\mathcal{A}$ contains a *fair cycle*: a cycle that is reachable from an initial state and intersects all the fair sets.

A Strongly-Connected Component (SCC) C of an automaton $\mathcal{A}$ is a maximal set of nodes such that there is a directed path between any node in C to any other. A reachable SCC that intersects all fair sets is called *fair* SCC. A SCC that intersects some initial states is called *initial* SCC. Given an automaton $\mathcal{A}$, the SCC (quotient) graph $Q(\mathcal{A})$ is the result of contracting each SCC of $\mathcal{A}$ into one node, merging the parallel edges and removing the self-loops.

Definition 2. *The* SCC (quotient) graph *of the automaton $\mathcal{A}$ is a four-tuple*

$$Q(\mathcal{A}) = \langle S^C, S_0^C, T^C, S_{\mathcal{F}}^C \rangle \ ,$$

where S^C is the finite set of SCCs, $S_0^C \subseteq S^C$ is the set of initial SCCs, $T^C = \{(C_1, C_2) | s_1 \in C_1, s_2 \in C_2 \ \text{ and } \ (s_1, s_2) \in T \ \text{ and } \ C_1 \neq C_2\}$ is the transition relation, $S_{\mathcal{F}}^C \subseteq S^C$ is the set of fair SCCs.

The SCC graph forms a Directed Acyclic Graph (DAG), which induces a partial order: The minimal (maximal) SCC has no incoming (outgoing) edges.

In symbolic model checking, we assume that all automata are defined over the same state space, agree on the state labels, and communicate through the common state space. The composition $\mathcal{A}_1 * \mathcal{A}_2 = \langle S, S_0, T, \mathcal{F}, A, \mathcal{L} \rangle$ of two Büchi automata $\mathcal{A}_1 = \langle S, S_{01}, T_1, \mathcal{F}_1, A, \mathcal{L} \rangle$ and $\mathcal{A}_2 = \langle S, S_{02}, T_2, \mathcal{F}_2, A, \mathcal{L} \rangle$ is defined by $S_0 = S_{01} \cap S_{02}$, $T = T_1 \cap T_2$, and $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$. Hence, composing two automata restricts the transition relation and results in the intersection of the two languages.

We also define a *quotient restriction* operation as follows:

[1] When the context is clear, we will just use $*$ to denote the composition operation between two FSMs. Similarly, consistent with BDD usage, we will sometimes use $*$ in place of $\times$ to refer to the cartesian product of two sets, or the product/composition of two automata.

Definition 3. *The restriction of* $\mathcal{A} = \langle S, S_0, T, \mathcal{F}, A, \mathcal{L} \rangle$ *by a subset SCC graph* $Q^- = \langle S^C, S_0^C, T^C, S_{\mathcal{F}}^C \rangle$ *is defined as* $\mathcal{A} \Downarrow Q^- = \langle S^-, S_0^-, T^-, \mathcal{F}, A, \mathcal{L} \rangle$, *with* $S^- = \{s | s \in C \text{ and } C \in S^C\}$, $S_0^- = \{s_0 | s_0 \in C_0 \text{ and } C_0 \in S_0^C\}$, $T^- = \{(s_1, s_2) | s_1, s_2 \in C \text{ and } C \in S^C \text{ and } (s_1, s_2) \in T\}$.

It follows that $\mathcal{A} \Downarrow Q(\mathcal{A}) = \mathcal{A}$. Unlike BDD restriction operations, the right argument here is a segment of a quotient graph. Inside the definition, the automaton is actually operated upon by the sets of states implied by the quotient graph.

An *SCC-closed set* of $\mathcal{A}$ is a subset $V \subseteq S$ such that, for every SCC C in $\mathcal{A}$, either $C \subseteq V$ or $C \cap V = \emptyset$. Note that if C is an SCC in $\mathcal{A}_1$ (or $\mathcal{A}_2$) , it is an SCC-closed set of the composition $\mathcal{A}_1 * \mathcal{A}_2$.

3 State Space Decomposition - Theory

The automaton $\mathcal{A}$ contains an accepting cycle iff its SCC graph $Q(\mathcal{A})$ contains a fair SCC.

Definition 4. *An SCC graph* $Q(\mathcal{A})$ *is "pruned" if all the minimal nodes are initial , all the maximal nodes are fair, and all the other nodes are on paths from initial nodes to fair nodes.*

Pruning (defined as removing nodes that are not in the pruned SCC graph $Q(\mathcal{A})$) does not change the language of the corresponding automaton $\mathcal{A}$. In the following, we assume that all the SCC graphs are pruned. [2]

The entire state space of $\mathcal{A}$ can be decomposed into state subspaces according to the structure of $Q(\mathcal{A})$. For brevity, we donot give proof for the following theorems since they are obvious.

Definition 5. *For each* fair *SCC* C_i *in* $Q(\mathcal{A})$, *we can construct an SCC subgraph* Q_i^F *by marking all the other SCCs "non-fair" and then pruning* $Q(\mathcal{A})$.

Theorem 1. *The language accepted by each state subspace* $\mathcal{A} \Downarrow Q_i^F$ *and the language accepted by* $\mathcal{A}$ *satisfy the following relations,*

$$L(\mathcal{A} \Downarrow Q_i^F) \subseteq L(\mathcal{A})$$

$$\cup_i L(\mathcal{A} \Downarrow Q_i^F) = L(\mathcal{A})$$

Note that in each SCC subgraph Q_i^F, the (only) maximal node is fair.

Definition 6. *In the SCC subgraph* Q_i^F, *each "initial-fair" path constitutes an SCC subgraph* Q_{ij}^L.

[2] In the *pruned* SCC graph, all the maximal nodes are fair. However, the *fair* SCCs are not always maximal – they might be on the paths from initial SCCs to other *maximal fair* SCCs.

Theorem 2. *The language accepted by each state subspace $\mathcal{A} \Downarrow Q_{ij}^L$ satisfies the following relations,*

$$L(\mathcal{A} \Downarrow Q_{ij}^L) \subseteq L(\mathcal{A} \Downarrow Q_i^F)$$

$$\cup_j L(\mathcal{A} \Downarrow Q_{ij}^L) = L(\mathcal{A} \Downarrow Q_i^F)$$

Therefore, the language emptiness of the original automaton $\mathcal{A}$ can be checked on each individual subgraphs $\mathcal{A} \Downarrow Q_{ij}^L$ separately.

Theorem 3. *$L(\mathcal{A}) = \emptyset$ iff $L(\mathcal{A} \Downarrow Q_{ij}^L) = \emptyset$ for every SCC subgraph Q_{ij}^L.*

In order to clarify the distinction between Cartesian product and composition operations in the sequel (see methods (b) and (c) in Section 4.3), we also include the following proposition.

Proposition 1. *Let $\{C_i^1\}$ be the SCCs of $\mathcal{A}^1$ and $\{C_j^2\}$ be the SCCs of $\mathcal{A}^2$. Then, the SCCs $\{C_{ij}\}$ of the composition $\mathcal{A}^1 * \mathcal{A}^2$ satisfies*

$$\begin{array}{ll} \exists_{k,l} \textit{ such that} & \\ (1) & C_{ij} \subseteq C_k^1 \times C_l^2, \\ (2) & C_{ij} * (C_{k'}^1 \times C_{l'}^2) = \emptyset, \forall (k',l') \neq (k,l), \end{array}$$

with equality holding only when the edges inside C_i^1 and C_j^2 either:

1. *have no labels; or*
2. *have labels whose supports are disjoint from each other; or*
3. *have mutually consistent labels (meaning nonempty conjunction).*

Although the first two conditions for equality are subsumed by the third, they demonstrate cheap tests that might be used to avoid the expensive composition operation in some cases.

4 The Algorithm

4.1 The Overall Algorithm

In this algorithm, we combine the idea of "sharp" guided search (will be explained in Section 4.4) with the "disjunctive" decomposition (explained in Section 3). The pseudo code of the overall algorithm is given in Figure 1. CHECK-LANGUAGE-EMPTINESS is the main procedure, which accepts three parameters: the concrete system $\mathcal{A}$, the property automaton $\mathcal{A}_{\neg\psi}$, and the list of (circuit) model submodules $\mathcal{M} = \{M_1, M_2, ..., M_m\}$. Our algorithm goes through the following phases:

1. The amassing phase: the property automaton $\mathcal{A}_{\neg\psi}$ is composed with submodules from $\{M_i\}$, one at a time, and its SCC graph $Q_{\mathcal{A}^+}$ is built at each step. This phase continues until either $Q_{\mathcal{A}^+}$ becomes an empty graph or the *amassing threshold* is reached. We will explain the amassing phase in detail in Section 4.2.

2. The decomposition and pre-jump phase: each fair SCC in $\mathcal{A}^+$ is pre-processed by being intersected with the remaining submodules in $\{M_i\}$. The details of this pre-jump process will be explained in Section 4.3. By building the QF/QL subgraphs, we decompose $Q_{\mathcal{A}^+}$ into the SCC subgraphs Q^F/Q^L.
3. The jump phase: now we "jump" to the concrete system $\mathcal{A}$, with the SCC subgraphs Q^L. Language emptiness is check on each individual state subspace $\mathcal{A} \Downarrow Q^L$. The "sharp" guided search idea is implemented in SHARPSEARCH-AND-LOCKSTEP, together with the LOCKSTEP search, with focus on the ideal goal of "early termination". This will be described in detail in Section 4.4.

4.2 Amassing and Decomposition

Amassing the Property Automaton The property automaton $\mathcal{A}_{\neg\psi}$ is usually small and its SCC graph usually demonstrates limited structure/sparsity. In order to get finer decomposition, we need to augment $\mathcal{A}_{\neg\psi}$ with a small portion of the submodules of $\mathcal{M} = \prod_i M_i$. At the very beginning, let the amassed automaton $\mathcal{A}^+ = \mathcal{A}_{\neg\psi}$. As we pick up the M_i and gradually add them to $\mathcal{A}^+$, we are able to see the structural interaction between the property automaton and the model. As a consequence, the SCCs in $\mathcal{A}^+$ gradually get fractured and the SCC graph becomes larger and shows more structure/sparsity. We call this augmentation process "amassing the automaton."

The order in which the remaining submodules M_i are brought in is critical, as is the way in which the original model was partitioned to form the submodules in the first place. Since our "sharpness" goal is to fracture the SCC graph and make it show more structure/sparsity, we have used the following criteria:

1. *Cone-Of-Influence* reduction: only state variables that are in the transitive fan-ins of $\mathcal{A}_{\neg\psi}$ are considered. These state variables are grouped into clusters $\{M_i\}$ so that the interaction between clusters is minimized [4]. For each cluster , we compute the SCC graph $Q(\mathcal{A}_i)$, with $\mathcal{A}_i = \mathcal{A}_{\neg\psi} * M_i$.
2. When we augment $\mathcal{A}^+$, we give priority to clusters that are both in the immediate fan-ins of $\mathcal{A}^+$ and have the relatively most complex SCC graph $Q(\mathcal{A}_i)$.
3. We repeat the previous step until either all the M_i are added, or the amassing phase reaches a certain threshold.

At each amassing step, current $\mathcal{A}^+$ is a *refinement* of the previous $\mathcal{A}^+$ (The SCC graph $Q(\mathcal{A}^+)$ is also a refinement of its previous counter-part). This means that we can build the SCC graph incrementally, as opposed to building it from the scratch every time. We use LOCKSTEP to refine each SCC of the previous $Q(\mathcal{A}^+)$, then update the edges. In addition, those SCCs that are in the previous SCC graph but are redundant (e.g. not in the pruned graph) are removed. If at anytime, $Q(\mathcal{A}^+)$ becomes empty, we can stop, knowing that the language is empty.

In order to avoid an excessive partitioning cost, with the consequent exponential number of subgraphs, we have a heuristic control on the activation of SCC refinement:

1. If the size of an SCC in the previous $Q(\mathcal{A}^+)$ is below a certain threshold, and it is not fair, do not refine it.

```
CHECK-LANGUAGE-EMPTINESS(A, A_¬ψ, {M_i}){        // entire system, property, submodules
    Reach := COMPUTE-INITIAL-STATES(A)
    A+ := A_¬ψ                                    // amassing phase
    while (amassing threshold not reached) do
        M_i := PICK-NEXT-SUBMODULE(A+, {M_i})
        A+ := A+ * M_i
        Q_A+ := BUILD-SCCGRAPH(A+)
        if Q_A+ is an empty graph then
            return true
        fi
    od
    for each fair SCC C ∈ Q_A+ do                 // decompose and pre-jump
        Q^F := BUILD-QF-SUBGRAPH(Q_A+, C)
        Queue := {C}
        for each remaining submodule M_i do
            Queue := REFINE-SCCS(A_¬ψ * M_i, Queue)
        od
        for each dfs path p_j in Q^F do
            Q^L := BUILD-QL-SUBGRAPH(Q^F, p_j)
            if (SHARPSEARCH-AND-LOCKSTEP(A, Q^L, C, Reach, Queue) = false) then
                return false
            fi
        od
    od
    return true
}

SHARPSEARCH-AND-LOCKSTEP(A, Q^L, C, Reach, Queue){   // model, hyper-line, fair SCC
                                                     // reachable, and SCC queue
    Front := Reach
    absRings := COMPUTE-REACHABLE-ONIONRINGS(A+ ⇓ Q^L)    // (see Definition 3)
FS: while (Front ≠ ∅) and (Front ∩ Queue = ∅) do
        Front := IMG#(A ⇓ Q^L, Front, absRings) \ Reach
        if (Front = ∅) then
            Front := IMG(A ⇓ Q^L, Reach) \ Reach
        fi
        Reach := Reach ∪ Front
    od
    if (Front = ∅) then
        return true
    else if (LOCKSTEP-WITH-EARLYTERMINATION(A ⇓ C, Queue, absRings)) then
        return false
    else
        goto FS
    fi
}
```

Fig. 1. The overall algorithm for checking language emptiness.

2. If the total number of edges in $Q(\mathcal{A}^+)$, e, exceeds a certain threshold, stop the *amassing*.
3. If the total number of fair SCCs in $Q(\mathcal{A}^+)$, f, exceeds a certain threshold, stop the *amassing*.

After the amassing phase, the SCC graph $Q(\mathcal{A}^+)$ is available. SCC subgraphs Q^F_i and Q^L_{ij} will be built as discussed in Section 3. Since each Q^L_{ij} corresponds to a depth-first search path in the SCC graph $Q(\mathcal{A}^+)$, we also called them *hyperlines* in the sequel. In fact, each hyperline is an envelope of Abstract Counter-Examples (ACE).

The total number of SCC subgraphs is bounded by the size of $Q(\mathcal{A}^+)$.

Theorem 4. *For an SCC graph with f fair SCCs and e edges, the total number of Q^F_i SCC subgraphs is f; The total number of the Q^L_{ij} SCC subgraphs is $O(fe)$.*

Let η_k denote the total number of states in $\mathcal{A}^+$. Without the control by the *amassing threshold*, $e = O(\eta_k^2)$ and $f = O(\eta_k)$ in the worst case. In our method, however, the amassing threshold bounds both f and e to constant values.

4.3 The Jump Phase

In the jump phase we determine whether any of the abstract counter examples in the current hyperlines contain a concrete counter example. We are currently using an intermediate iterative state subspace restriction process that can be inserted before "jump", which is related to the work of [4], and [5].

Assume that the submodules of the model ($\mathcal{M} = \prod_i M_i$) have each been intersected with the property automaton, creating a series of $\mathcal{A}_i$, where $i = 1, 2, ..., m$ [3]. After the amassing phase, we have: (1) the *amassed* automaton $\mathcal{A}^+ = \mathcal{A}_1 * \mathcal{A}_2 * ... * \mathcal{A}_{k-1}$, (2) the remaining automata $\mathcal{A}_k, \mathcal{A}_{k+1}, ..., \mathcal{A}_m$, and (3) the list of SCC-closed sets in $\mathcal{A}^+$, which we shall call L^+.

At this point, LOCKSTEP can be used to partition and refine each SCC-closed set of L^+ into one or more SCCs according to the Transition Relations (TR) of $\mathcal{A}_i$. We briefly discuss four different approaches to the "jump" phase. The first is used for a similar purpose in D'n'C, while the last three are part of our new algorithm.

In the last three approaches, the last step, called the jump step, is the same: we search for fair cycles on the concrete system, subject to a computed state subspace restriction. Only the state subspace restriction varies from method to method.

First, the D'n'C approach [16], which we shall call Method (a), is described as follows:

$$L = \mathrm{EL}(\prod_i \mathcal{A}_i, \cup_{C \in L^+} C) \ ,$$

where EL stands for the *Emerson-Lei* algorithm, and $\cup_{C \in L^+} C$ is the union of all the fair SCC-closed sets of $\mathcal{A}^+$. $\prod_i \mathcal{A}_i$ is the concrete system. Its main feature is that the fair cycle detection is restricted to each state subspace $C \in L^+$. In [16], the advantageous experimental results were attributed mainly to this restriction and the automata strength reduction [2].

[3] $\mathcal{A}_i = \mathcal{A}_{\neg\psi} * M_i$, and $\mathcal{A} = \prod_i \mathcal{A}_i$

The second approach, which is the one currently in our implemention, can be called the "Cartesian product" approach, and will be referred to as Method (b). It is based directly on Proposition 1, and can be characterized as follows: [4]

Compute Jump State Space Restriction	
L_k	$= \text{LOCKSTEP}(\mathcal{A}_k, L^+)$
L_{k+1}	$= \text{LOCKSTEP}(\mathcal{A}_{k+1}, L_k)$
L_{k+2}	$= \text{LOCKSTEP}(\mathcal{A}_{k+2}, L_{k+1})$
...	
L_m	$= \text{LOCKSTEP}(\mathcal{A}_m, L_{m-1})$
Jump in Restricted State Space	
L	$= \text{LOCKSTEP}(\prod_i \mathcal{A}_i, L_m)$

A direct analogy can be observed between Method (b) and the MBM (Machine by Machine) approach of [4] in computing the approximate reachable states..

Each SCC-closed set C in list L_k is further partitioned by LOCKSTEP into a collection of SCCs according to the TR of $\mathcal{A}_{k+1}$. The submachines remaining to be composed in testing the ACE are treated "one machine at a time." Note that the quotient graph of machine $\mathcal{A}_k = \mathcal{A}_{\neg\psi} * M_k$ has been computed a priori, and the searches inside LOCKSTEP are restricted to the state subspace $C' \times C$, where C' is an SCC of $\mathcal{A}_k$, and $\times$ represents the Cartesian product.

The product $C' \times C$ is usually a smaller set than C, because the product operation further refines the partition block C. In addition, this process can fracture these SCC-closed sets, for C and C' are sometimes disjoint sets. Thus, sets in L_k can be smaller than sets in L^+. Similarly, sets in L_{k+1} can be smaller still, and so on. As the machine that LOCKSTEP operates on becomes progressively more concrete, the size of the considered state space becomes progressively smaller. Therefore, the state restriction by Method (b) is generally much tighter than in Method (a).

To illustrate this effect, consider a simple example in which the property automaton $\mathcal{A}_{\neg\psi}$ has two fair SCCs (scc1, scc2), and the pre-jump amassed automaton $\mathcal{A}^+$ also has two fair SCCs (SC1, SC2). We assume: scc1 $\supseteq$ SC1 and scc1 $\times$ SC2 $= \emptyset$. Suppose that there are two submodules M_a, M_b yet to be composed, M_a has a single fair SCC C_a, and M_b also has a single fair SCC C_b. Summarizing, we have

Module	Fair SCCs	
$\mathcal{A}_{\neg\psi}$	scc1	scc2
$\mathcal{A}^+$	SC1	SC2
$\mathcal{A}_a = \mathcal{A}_{\neg\psi} * M_a$	C1	C2
$\mathcal{A}_b = \mathcal{A}_{\neg\psi} * M_b$	C3	C4
M_a	C_a	
M_b	C_b	

After the composition $\mathcal{A}_{\neg\psi} * M_a$, C_a is decomposed into two SCCs (C1,C2). In this case, it is obvious that $C1 \subseteq (scc1 \times C_a)$ and $C2 \subseteq (scc2 \times C_a)$. The same thing happens to the composition of M_b: its only fair SCC C_b is decomposed into (C3, C4). Therefore, we have $C3 \subseteq (scc1 \times C_b)$, $C4 \subseteq (scc2 \times C_b)$.

[4] In the pseudo code, this is described by the function REFINE-SCCS.

We take the two cartesian products to yield

$$\begin{array}{ll} \textsc{LockStep}(M_a, \{SC1, SC2\}) & = \{SC1 \times C_a, SC2 \times C_a\} \\ \textsc{LockStep}(M_b, \{SC1 \times C_a, SC2 \times C_a\}) & = \{SC1 \times C_a \times C_b, SC2 \times C_a \times C_b\} \end{array}$$

Notice that

$$\begin{array}{l} SC1 \times C_a \times C_b \supseteq SC1 \times (C1 + C2) \times (C3 + C4) \\ SC2 \times C_a \times C_b \supseteq SC2 \times (C1 + C2) \times (C3 + C4) \end{array}$$

To Summarize,

$$\begin{array}{lll} L^+ = & \{SC1, SC2\} & \text{amassing} \\ L_a = \textsc{LockStep}(\mathcal{A}_a, L^+) & = \{SC1 \times C1, SC2 \times C2\} & \text{refining } \mathcal{A}_a \\ L_b = \textsc{LockStep}(\mathcal{A}_b, L_a) & = \{SC1 \times C1 \times C3, SC2 \times C2 \times C4\} & \text{refining } \mathcal{A}_b \\ L = \textsc{LockStep}(\prod_i \mathcal{A}_i, L_b) & \text{(Jump in Restricted State Space)} & \end{array}$$

Since $SC1 \subseteq scc1$, $SC2 \subseteq scc2$, Method (b) gives a smaller restriction subspace than Method (a) as used in D'n'C.

The third approach, Method (c), can be called the "one-step composition" approach, and can be characterized briefly as follows:

$$\begin{array}{ll} L_k & = \textsc{LockStep}(\mathcal{A}^+ * \mathcal{A}_k, L^+) \\ L_{k+1} & = \textsc{LockStep}(\mathcal{A}^+ * \mathcal{A}_{k+1}, L_k) \\ \ldots & \\ L_m & = \textsc{LockStep}(\mathcal{A}^+ * \mathcal{A}_m, L_{m-1}) \\ & \\ L & = \textsc{LockStep}(\prod_i \mathcal{A}_i, L_m) \end{array}$$

Whereas Method (b) does no composition prior to making the full jump, Method (c) invests more heavily in sharpness by composing $\mathcal{A}^+$ with each of the remaining submodules. At each step, we use the refined SCC-closed sets computed in the previous step. There is certainly more work than in method (b), but it produces still "sharper" (that is, smaller) restriction subspaces, due to the SCC-fracturing process inherent in composition. In comparing Methods (b) and (c), the reader should pay attention to Proposition 1. Working with unlabeled graphs might give the impression that methods (b) and (c) give identical results.

For an edge to exist in the STG of the composition, it must exist in both of the machines being composed. Whereas method (b) never "fractured" any individual SCCs, Method (c) does, ultimately leading to much smaller restriction subspaces in the jump (that is the last) step.

The fourth approach, Method (d), can be called the "full iterative composition" approach, and can be characterized as follows:

$$\begin{array}{ll} L_k & = \textsc{LockStep}(\mathcal{A}^+ * \mathcal{A}_k, L^+) \\ L_{k+1} & = \textsc{LockStep}((\mathcal{A}^+ * \mathcal{A}_k) * \mathcal{A}_{k+1}, L_k) \\ L_{k+2} & = \textsc{LockStep}(((\mathcal{A}^+ * \mathcal{A}_k) * \mathcal{A}_{k+1}) * \mathcal{A}_{k+2}, L_{k+1}) \\ \ldots & \\ L & = \textsc{LockStep}(\prod_i \mathcal{A}_i, L_m) \end{array}$$

In the calls to LOCKSTEP, the next of remaining submachine is composed with the result of the previous composition. At each step, computation is restricted to the SCC-closed sets computed in the previous step.

This composition process maximally fractures the SCC closed sets. Each step is thus done on a maximally reduced restriction subspace, due to the restriction to the state subspace of an SCC computed in the previous step. Furthermore, the SCCs of L_{k+1} are generally smaller than those in L_k. As the machine on which LOCKSTEP operates becomes progressively more concrete, the size of the considered state space becomes progressively smaller. Method (d) is offered to complete the spectrum of available sharpness options. It has not yet been implemented.

The principle at work in Methods (a)-(d) is to use the maximum affordable sharpness in each composition step. Method (a) represents the least investment in sharpness, and therefore suffers the least amount of overhead. However, it performs the most expensive step (the jump step) on the largest subspace. On the other hand, Method (d) is the sharpest at the jump step, but incurs the greatest overhead.

Roughly speaking, we expect that

$$\text{CPUTIME(a)}) \ll \text{CPUTIME(b)} \ll \text{CPUTIME(c)}) \ll \text{CPUTIME(d)}$$

However, in the experimental results section we show that the largest computations are only possible with maximum affordable sharpness. The larger investment is clearly justified when the cheaper approach fails anyway.

4.4 Sharp Search and Fair Cycle Detection

After jumping to the concrete system, we need to check the language emptiness on the individual state subspace ($\mathcal{A} \Downarrow Q_{ij}^{L_i}$). Fortunately, the subspaces are much smaller than the entire state space, and as a result, both reachability analysis and fair cycle detection are easier.

Since fair cycle detection is generally harder than the forward search (reachability analysis), and it does not make sense to search in the unreachable area for a fair cycle, we want to do the forward search first, only starting fair cycle detection when the forward search hits a *promising* state. Promising states are defined as the states that are in the SCC-closed sets of $\mathcal{A}^+$ and at the same time satisfy some fairness constraints. Promising states are also prioritized based on the number of fairness constraints they satisfy: those that satisfy more get higher priorities.

Sharp Search Even after the disjunctive decomposition, some hyperlines may not be as "sharp" as expected. This is because the SCC size varies, and there may be a big SCC stay in the hyperline. In such cases, we need to sharpen the forward search further. To address this problem, we present in the following our "sharp" guided search algorithm.

Instead of using the normal image computation, at each step of the forward search, we use its "sharp" counterpart – IMG$^{\#}$. The pseudo code of IMG$^{\#}$ is given in Fig. 2. First, a subset of the "from" set is computed heuristically, (it could be a minterm, a cube , or an arbitrary subset with a small BDD representation), and states in this subset is selected in such a way that those with shorter approximate distances to the fair SCCs are favored.

IMG$^{\#}$ is fast even on the concrete system; furthermore, it heuristically targets the fair SCCs. Therefore, it is able to find a fair SCC by visiting only part of the states in the *stem* (states between initial states and fair SCCs).

```
IMG#(A, From, absRings){        // Model, from set, and abstract onionRings
    i := LENGTH(absRings)
    while (From ∩ absRings[i] = ∅) do
        i − −
    od
    From# := BDD-SUBSETTING(From ∩ absRings[i])
    return IMG(A, From#)
}
```

Fig. 2. The "sharp" image computation algorithm

Since IMG$^{\#}$ computes only a subset of the normal image, a *dead-end* might be reached (e.g. its result is an empty set) before the forward search reaches the fix-point. Whenever this happens, we need to *backtrack* and use the normal IMG to recover (this is described in Figure 1).

If there exist fair cycles, the sharp guided search algorithm might be able to find one by exploring only part of the reachable states and going directly to its target - the fair SCCs. However, all the reachable states or all the SCC-closed sets (whichever finishes first) must be explored if there is no fair cycle. In the worst case, the sharp search have to be executed on every hyperline. Since some area (states) may be shared by more than one hyperlines, we use the variable *Reach* (Figure 1) to avoid traversing them more than once.

Let η_R be the total number of *reachable* state, and fe be the number of hyperlines (or Q^L SCC subgraphs), the total cost of the sharp guided search is $O(\eta_R + fe)$.

Prioritized Lockstep with Early Termination LOCKSTEP with early termination is used together with sharp guided search to search for fair cycles on $\mathcal{A} \Downarrow Q^L_{ij}$.

All the SCC-closed sets are put into a *priority queue*, where they are prioritized according to the approximate distances to the initial states. (These distances are computed on the abstract model $\mathcal{A}^+$, based on the abstract reachable onion rings). The recursion in LOCKSTEP is also implemented by a priority queue [13]. When the forward search hits some promising states, one of them (with a higher priority) is selected as a *seed* to start LOCKSTEP. This guarantees that every fair SCC found is in the reachable area.

The early termination is implemented as follows: at each symbolic step inside LOCKSTEP, we check whether the intersection of the states reached forwardly and backwardly from the seed satisfy the fairness conditions. The procedure terminates as soon as the intersection satisfies all fairness conditions, because it contains fair cycles that are both reachable and fair.

Assume that η is the total number of states on the concrete system, the cost of the fair cycle detection with LOCKSTEP is bounded by $O(\eta \log \eta)$.

4.5 Complexity

Let $\mathcal{A}_1^+, \mathcal{A}_2^+, ..., \mathcal{A}_k^+$ be the series of over-approximated abstract models in the amassing phase. Assume that $\mathcal{A}$ contains r binary state variables, the total number of states is $\eta = O(2^r)$. Since the abstract models and the concrete system are defined over the same state space and agree on the state labels, each abstract model also has η states. However, the *effective number of states* in an abstract model is significantly smaller: for $\mathcal{A}_i^+$ with $t_i \leq r$ binary state variables, its states can be partitioned into 2^{t_i} blocks, inside each of which states are "indistinguishable". Therefore, we define $\eta_i = O(2^{t_i})$ as the *effective number of states* of the abstract model. It follows that LOCKSTEP takes $O(\eta_i \log \eta_i)$ symbolic steps on $\mathcal{A}_i$.

Amassing and Decomposition Building the SCC quotient graph for $\mathcal{A}_i^+$ takes $O(\eta_i \log \eta_i)$ symbolic steps. For any two consecutive abstract models $\mathcal{A}_i^+$ and $\mathcal{A}_{i+1}^+$, the second one has at least one more binary state variable, which gives us the following relation over their *effective number of states*:

$$\frac{\eta_{i+1}}{\eta_i} \geq 2$$

Thus, the total cost of the amassing phase is bound by

$$\eta_k \log \eta_k (1 + 1/2 + 1/4 + ...) \leq 2\eta_k \log \eta_k \ ,$$

which is $O(\eta_k \log \eta_k)$. A similar argument can be applied to the pre-jump process.

During the decomposition phase, the total number of hyperlines is $O(fe)$, assuming that the SCC quotient graph of $\mathcal{A}_k^+$ has a total number of f fair SCCs and e edges.

Sharp Search and Lockstep In the worst case, SHARP-SEARCH visits all the reachable states, plus at least one extra image computation on every state subspace, which means the total cost is bounded by $O(\eta_R + fe)$. In additiona, fair cycle detection on the concrete system is bounded by $O(\eta \log \eta)$ symbolic steps.

Put all of them together, we have the overall complexity

$$O(\eta_k \log \eta_k + fe + \eta_R + \eta \log \eta + fe) = O(\eta \log \eta + fe) \ .$$

In our implementation, fe is bounded by a constant value (the amassing threshold). (Leaving it uncontrolled will result in $O(fe) = O(\eta_k^3)$ in the worst case .)

5 Experiments

We implemented our algorithm in VIS-1.4 (we call it LEC$^{\#}$), and compared its performance with both Emerson-Lei (the standard language emptiness checking command) and D'n'C on the circuits from [16] and the texas97 benchmark circuits. All experiments used the static variable orderings (obtained by the dynamic variable reordering command in VIS). Experiments in Table 1 and Table 2 were conducted on a 400MHz Pentium II with 1GB of RAM, while the harder experiments in Table 3 were on a 1.7

GHz Pentium 4 with 1GB of RAM. All of them were running Linux and with the data size limit set to 750MB.

Table 1 shows that with VIS dcLevel=2 (using prior reachability analysis results as don't cares where possible), D'n'C consistently outperforms our new algorithm. To summarize the comparison of the new algorithm and D'n'C, we can denote by "CL" (Constant factor Lose) the cases in which both algorithms complete, but D'n'C is faster. Similarly, we can denote by "CW" (Constant factor Win) the cases in which both algorithms complete, but LEC$^{\#}$ is faster. We also denote by "AL/AW" (Arbitrary Factor Loss/Win) the cases where D'n'C (the new algorithm) completes but the other doesn't. With this notation, a tally of Table 1 gives

Cases	LEC$^{\#}$ vs. D'n'C	LEC$^{\#}$ vs. EL
CL	15	6
CW	3	6
AL	0	0
AW	0	6

Compared to D'n'C, LEC$^{\#}$ has only 3 constant factor wins vs. 15 for D'n'C. However, among D'n'C's 15 CWs, only 4 are for problems needing more than 100 seconds to complete – that is, the easy problems. In contrast, on LEC$^{\#}$'s 3 CWs, D'n'C took 1337, 1683, and 233 seconds. Neither algorithm has an AW case. We conclude that on harder problems, LEC$^{\#}$ is at least competitive even when advance reachability is feasible.

A similar tally for LEC$^{\#}$ vs. EL shows that LEC$^{\#}$ ties with EL in the constant factor competition (with 6 CWs each), and has a convincing advantage in AWs: New 6, EL 0.

Tables 2 and 3 show that with VIS dcLevel=3 (using approximate reachability analysis results as don't cares), LEC$^{\#}$ consistently out-performs both D'n'C and EL on the circuits of [16] and the Texas-97 benchmark circuits. The difference here is that both D'n'C and EL depend strongly on full reachability [5] to restrict the search spaces; however, the sharp search of our new algorithm minimizes this dependency.

For the circuits of [16], the tally is:

Cases	LEC$^{\#}$ vs. D'n'C	LEC$^{\#}$ vs. EL
CL	6	3
CW	4	5
AL	0	0
AW	8	10

Compared to D'n'C, LEC$^{\#}$ has only 4 constant factor wins vs. 6 for D'n'C. However, all D'n'C's 6 CWs are for problems needing less than 100 seconds to complete. In contrast, on LEC$^{\#}$'s 4 CWs, D'n'C took 7565, 5, 2165, and 1139 seconds; except for one case, LEC$^{\#}$ "wins the big ones."

The bottom line that we are seeking is completion on large problems. In that respect, note that AWs (Arbitrary Factor Wins) are LEC$^{\#}$ 8, D'n'C 0.

A similar tally for LEC$^{\#}$ vs. EL shows that LEC$^{\#}$ wins the constant factor competition, and it has an even more convincing advantage in AWs: New 10, EL 0.

[5] The full reachability analysis is usually impossible on pratical circuits.

Table 1. On the circuits of [16]: * means dcLevel=0 (no don't cares), otherwise, dcLevel=2 (reachable don't cares). T/O means time-out after 4 hours.

			CPU (s)			Memory (MB)			BDD (M)		
Circuit and LTL	Pass or Fail	latch num	EL	D'n'C	LEC$^{\#}$	EL	D'n'C	LEC$^{\#}$	EL	Dnc	LEC$^{\#}$
bakery1	F	56	212	27	159	262	75	125	5.1	1.3	1.5
bakery2	P	49	69	20	28	152	73	74	3.4	1.2	1.2
bakery3	P	50	421	43	1514	550	111	125	14	1.8	1.5
bakery4	F	58	T/O	1337	655	-	411	476	-	4.7	4.8
bakery5	F	59	T/O	623	737	-	555	554	-	6.1	9.9
eisen1	F	35	23	16	128	69	50	64	1.0	0.9	0.6
eisen2	F	35	T/O	1683	944	-	564	340	-	7.7	1.7
elevator1	F	37	210	41	192	489	132	369	14	2.2	10.3
nmodem1	P	56	4384	233	227	569	63	169	11	0.6	2.2
peterson1	F	70	17	21	41	73	83	78	1.1	1.2	1.2
philo1	F	133	371	7	56	401	26	37	12	0.2	0.1
philo2	F	133	73	12	58	145	44	42	2.8	0.5	0.3
philo3	P	133	T/O	115	207	-	119	329	-	1.2	7.0
shamp1	F	143	44	87	303	96	113	401	2.1	2.2	9.2
shamp2	F	144	T/O	101	239	-	187	268	-	2.9	3.5
shamp3	F	145	T/O	335	1383	-	478	500	-	4.4	5.8
twoq1*	P	69	12	4	14	36	23	24	0.4	0.1	0.0
twoq2*	P	69	241	30	289	333	47	509	8.9	0.9	7.9

Finally, we look at the same comparisons on the Texas-97 benchmark circuits. Similarly tallying Table 3, we obtain

Cases	LEC$^{\#}$ vs. D'n'C	LEC$^{\#}$ vs. EL
CL	2	1
CW	3	3
AL	0	0
AW	2	3

On these mostly larger circuits, for some of which reachability analysis is prohibitively expensive, we see a decisive advantage of LEC$^{\#}$ vs. both D'n'C and EL.

6 Conclusion

We have proposed a new language emptiness checking algorithm based on a series of "sharpness" heuristics, which enable us to perform the most expensive part of the computation in restricted state subspaces. We have presented both theoretical and experimental results, which support our hypothesis that for large or otherwise difficult problems, heavy investment in the state space decomposition is justified.

While D'n'C mostly is faster than our new algorithm on problems where prior reachability analysis is possible, the new algorithm outperforms both the Emerson-Lei

Table 2. On the circuits of [16] : With dcLevel=3 (approximate reachable don't cares). T/O means time-out after 4 hours.

			CPU			Memory			BDD		
Circuit and LTL	Pass or Fail	latch num	EL	D'n'C	LEC$^{\#}$	EL	D'n'C	LEC$^{\#}$	EL	Dnc	LEC$^{\#}$
bakery1	F	56	T/O	7565	5367	-	609	447	-	17.6	8.0
bakery2	P	49	183	5	2	241	25	15	4.1	0.1	0.0
bakery3	P	50	2794	48	174	609	128	133	18.8	2.1	1.5
bakery4	F	58	T/O	T/O	1964	-	-	477	-	-	4.0
bakery5	F	59	T/O	T/O	1294	-	-	416	-	-	4.9
eisen1	F	35	23	6	107	36	26	73	0.3	0.3	0.5
eisen2	F	35	T/O	T/O	1150	-	-	365	-	-	3.0
ele	F	37	3504	2156	585	663	612	657	24.4	21.1	23.6
nullmodem	P	56	T/O	T/O	3375	-	-	306	-	-	2.6
peterson	F	70	4	8	176	21	42	121	0.0	0.3	1.4
philo1	F	133	T/O	T/O	385	-	-	64	-	-	0.9
philo2	F	133	T/O	T/O	267	-	-	144	-	-	2.1
philo3	P	133	T/O	1139	241	-	609	119	-	21.4	1.4
shampoo1	F	143	12	T/O	168	21	-	127	0.0	-	2.0
shampoo2	F	144	T/O	T/O	189	-	-	153	-	-	3.0
shampoo3	F	145	T/O	53	735	-	51	331	-	0.3	5.0
twoq1	P	69	12	4	23	37	14	24	0.4	0.1	0.0
twoq2	P	69	172	30	665	322	15	496	7.7	0.9	8.2

algorithm and D'n'C on more difficult circuits. Although our new algorithm does not win in every case, it tends to win on the harder problems. Out of the 25 LTL model checking cases we described in Tables 2 and 3, Emerson-Lei timed out in 13 cases – more than half, which attests to the fact that the circuits studied, while not huge, are definitely non-trivial. The D'n'C algorithm timed out on 10 of the 25 cases. Since our algorithm never timed out (and usually had much smaller memory requirements when time was not an issue), we can only say that the speedup achieved in these cases was arbitrarily large.

We note that our new algorithm does not yet employ the strength reduction techniques of D'n'C. This suggests that sharpness itself is very powerful. However, when combined with the strength reduction techniques, our advantage with respect to both D'n'C and Emerson-Lei might improve further.

A priority in future work will be to diagnose the qualities of a given design that make language emptiness checking compute-intensive. This might afford guidance on how to set the various parameters of our algorithm, such as how many latches to compose before jumping, and how to choose between sharp forward search and sharp backward search at the end of the jump phase (currently, we start both and abandon the one that seems to be stalling). Further research will also be focused on the clustering algorithms to create the submodules, and on the corresponding refinement scheduling (guidance on the order of processing the submodules in the amassing and jump phases).

Table 3. On Texas-97 benchmark circuits. With dcLevel=3 (approximate reachable don't cares). T/O means time out after 8 hours.

			CPU (s)			Memory (MB)			BDD (M)		
Circuit and LTL	Pass or Fail	latch num	EL	D'n'C	LEC$^{\#}$	EL	D'n'C	LEC$^{\#}$	EL	D'n'C	LEC$^{\#}$
Blackjack1	F	176	7296	2566	237	618	610	551	26.8	24.2	18.1
MSI cache1	P	65	T/O	T/O	51	-	-	83	-	-	2.0
MSI cache2	F	65	T/O	T/O	165	-	-	342	-	-	6.7
PI bus1	P	387	T/O	73	1700	-	243	539	-	3.5	13.4
PI bus2	F	385	501	292	1302	467	477	609	17.0	15.4	22.6
PPC60X1	F	67	1109	1690	651	609	611	445	20.1	22.4	10.6
PPC60X2	P	69	13459	2811	531	745	625	327	17.8	18.9	6.9

7 Acknowledgements

We acknowledge the contributions of "deep in the shed" research sessions with Roderick Bloem, Kavita Ravi, and Fabio Somenzi.

References

[1] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, November 2000. LNCS 1954.

[2] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633.

[3] 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.

[4] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A state space decomposition algorithm for approximate FSM traversal. In *Proceedings of the European Conference on Design Automation*, pages 137–141, Paris, France, February 1994.

[5] D. L. Dill. What's between simulation and formal verification? In *Proceedings of the Design Automation Conference*, pages 328–329, San Francisco, CA, June 1998.

[6] 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.

[7] 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.

[8] 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.

[9] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

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

[11] 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*, pages 97–107, New Orleans, January 1985.

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

[13] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.

[14] 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.

[15] 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.

[16] C. Wang, R. Bloem, G. D. Hachtel, K. Ravi, and F. Somenzi. Divide and compose: SCC refinement for language emptiness. In *International Conference on Concurrency Theory (CONCUR01)*, pages 456–471, Berlin, August 2001. Springer-Verlag. LNCS 2154.

[17] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design*, 19(10):1225–1230, October 2000.