Divide and Compose: SCC Refinement for Language Emptiness[⋆]

Chao Wang[1], Roderick Bloem[1], Gary D. Hachtel[1], Kavita Ravi[2], and Fabio Somenzi[1]

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

Abstract. We propose a refinement approach to symbolic SCC analysis, which performs large parts of the computation on abstracted systems, and on small subsets of the state space. For language-emptiness checking, it quickly discards uninteresting parts of the state space; for the remaining states, it adapts the model checking computation to the strength of the SCCs at hand.
We present a general framework for SCC refinement, which uses a compositional approach to generate and refine overapproximations. We show that our algorithm significantly outperforms the one of Emerson and Lei.

1 Introduction

Checking language emptiness of a Büchi automaton is a core procedure in LTL [12, 17] and fair-CTL model checking [13], and in language-containment based verification approaches [11]. The classical algorithm by Emerson and Lei [7] used in symbolic model checkers is based on the computation of an *SCC hull* [15], while Xie and Beerel [18] and Bloem, Gabow, and Somenzi [1] use SCC decomposition to decide language emptiness.

Although the Lockstep algorithm of [1] has a better complexity than the one of Emerson and Lei ($n \log n$ versus n^2), the comparison presented in [15] shows that the theoretical advantage seldom translates in shorter CPU times. We present an algorithm that uses abstractions to compute an SCC decomposition of the system by refinement. It combines this with known language emptiness approaches to form a hybrid algorithm that shares the good theoretical characteristics of Lockstep, while outperforming the most popular SCC-hull methods, including the one of Emerson and Lei. This *Divide and Compose* algorithm, called *D‘n’C*, has the following features:

1. It is compositional, performing as much work as possible on abstracted systems.
2. It considers only parts of the state space at any time.
3. It uses the *strength* of a given set of SCCs in a given system to decide the proper model checking algorithm.

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

We assume that the model is the conjunction of a large number of modules, including the Büchi automaton for the property. Our approach exploits property locality [11] by first performing an SCC decomposition on an abstraction obtained by composing a small number of modules. Then it automatically refines the system by composing the current abstraction with one of the previously omitted modules, which enables it to refine the SCC decomposition in turn.

At any stage, if an SCC of an abstracted system does not contain a fair cycle, then we can safely discard that part of the state space, which means we do not have to consider it in a more refined system. Because each SCC of a system is contained in an SCC of a more abstract system, and because we do not have to consider all SCCs, we can often drastically limit the potential space in which a fair cycle can lie. This allows us to make very efficient use of don't care conditions.

The *strength* of a Büchi automaton [10, 2] is an important factor in symbolic model checking. Specialized model checking algorithms for weak and especially terminal automata outperform the general language emptiness algorithm of Emerson and Lei: $\mathsf{EF}\,\mathsf{EG}\,\mathsf{fair}$ can be used for weak systems and $\mathsf{EF}\,\mathsf{fair}$ can be used for terminal ones. For strong automata, however, a general fair cycle detection algorithm must be used.

The classification of [2] determines which model checking approach to use, based on the strength of the Büchi *automaton*. This may be inefficient because a Büchi automaton that contains one strong SCC and several weak ones is classified as strong. Our approach considers the strength of each *individual SCC* to decide which model checking procedure to use. Furthermore, it uses *strength reduction*: the fact that the composition of a strong SCC with a Kripke structure may contain weak SCCs. Our approach analyzes SCCs as they are computed to take maximal advantage of their weakness.

The rest of this paper is organized as follows. Section 2 reviews the background material. Section 4 discusses the algorithmic framework for SCC refinement, while Section 5 deals with the compositional approach. Section 6 discusses the implementation details, and presents preliminary experimental results. These results show that the algorithm often achieves substantial savings in memory and CPU time. Section 7 summarizes the contributions of the paper and outlines promising future work.

2 Preliminaries

A *Strongly-Connected Component (SCC)* C of a graph G is a maximal set of nodes such that there is a path between any two nodes in C. An SCC that consists of just one node without a self loop is called *trivial*. An *SCC-closed set* of G is the union of a collection of SCCs. The set of SCCs of G is denoted by $\mathrm{SCCs}(G)$ and is a partition of the vertices of G. A partition π_1 of V is a *refinement* of another partition π_2 of V if, for every $B_1 \in \pi_1$, there exists $B_2 \in \pi_2$ such that $B_1 \subseteq B_2$.

We model the systems we consider as generalized Büchi automata.

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

$$\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle \enspace ,$$

where Q is the finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $T \subseteq Q \times Q$ is the transition relation, $\mathcal{F} \subseteq 2^Q$ is the set of acceptance conditions, A is the finite set of atomic propositions, and $\Lambda : Q \to 2^A$ is the labeling function.

Note that we have defined automata with labels on the states, not the edges. A *run* of $\mathcal{A}$ is an infinite sequence $\rho = \rho_0, \rho_1, \ldots$ over Q, such that $\rho_0 \in Q_0$, and for all $i \geq 0$, $(\rho_i, \rho_{i+1}) \in T$. A run ρ is *accepting* (or *fair*) if, for each $F_i \in \mathcal{F}$, there exists $q_j \in F_i$ 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 $i \geq 0$, $\sigma_i \in \Lambda(\rho_i)$. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the subset of A^ω accepted by $\mathcal{A}$. The language of $\mathcal{A}$ is nonempty iff $\mathcal{A}$ contains an *accepting cycle*: a cycle that is reachable from an initial state and intersects all acceptance conditions. An automaton contains an accepting cycle iff it contains an *accepting SCC*: a reachable SCC that intersects all accepting sets.

A state q is *complete* if for every $a \in A$ there is a successor q' of q such that $a \in \Lambda(q')$. A set of states, or an automaton, is complete if all of its states are.

As is usual in symbolic model checking, given a transition relation T, we define $\mathrm{img}_T(S) = \{q' \mid \exists q \in S : (q, q') \in T\}$, and $\mathrm{pre}_T(S) = \{q \mid \exists q' \in S : (q, q') \in T\}$. A *step* is the computation of either $\mathrm{img}_T(S)$ or $\mathrm{pre}_T(S)$.

We assume that all automata are defined over the same state space and agree on the state labels. Communication then proceeds through the common state space, and composition is characterized by the intersection of the transition relations: The composition $\mathcal{A}_1 \times \mathcal{A}_2 = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$ of two Büchi automata $\mathcal{A}_1 = \langle Q, Q_{01}, T_1, \mathcal{F}_1, A, \Lambda \rangle$ and $\mathcal{A}_2 = \langle Q, Q_{02}, T_2, \mathcal{F}_2, A, \Lambda \rangle$ is defined by $Q_0 = Q_{01} \cap Q_{02}$, $T = T_1 \cap T_2$, 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.

If the states of a Büchi automaton are the valuations of r binary state variables, then language emptiness can be checked by a symbolic algorithm in $O(|Q| \log |Q|) = O(r2^r)$ steps [1]. We can improve this bound if we know that the automaton does not control some of the state variables. Specifically, let V a finite set (of binary variables), and let $\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$ be an automaton such that $Q = 2^V$. Given $q \in Q$ and $v \in V$, let $q^v \in Q$ be the state given by $q \cup \{v\}$ if $v \notin q$ and $q \setminus \{v\}$ otherwise. Then $\mathcal{A}$ *controls* v if there exist $q_1, q_2 \in Q$ such that $(q_1, q_2) \in T$ and $(q_1, q_2^v) \notin T$. Let $V_\mathcal{A}$ the set of variables controlled by $\mathcal{A}$. We define the *effective* number of states of $\mathcal{A}$ as $\eta_\mathcal{A} = 2^{|V_\mathcal{A}|}$. One can easily show that language emptiness for $\mathcal{A}$ can be checked in $O(\eta_\mathcal{A} \log \eta_\mathcal{A})$ steps by the algorithm of [1].

We say that $\mathcal{A} \leq \mathcal{A}'$ if $Q = Q'$, $Q_0 \subseteq Q_0'$, $T \subseteq T'$, $\mathcal{F} = \mathcal{F}'$, and $\Lambda = \Lambda'$. This (rather strong) condition induces a partial order on automata, such that $\mathcal{A} \leq \mathcal{A}'$ implies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. If $\mathcal{A} \leq \mathcal{A}'$, we say that $\mathcal{A}'$ is an *overapproximation* of $\mathcal{A}$.

Let $C \subseteq Q$ be an SCC of $\mathcal{A}$. We define the *strength* of C as follows (cf. [10, 2]).

– C is *weak* if all cycles contained within it are accepting.
– C is *terminal* if it is weak, complete, and there is no edge from a node in C to any non-terminal SCC. Terminality implies acceptance of all runs reaching C.
– C is *strong* if it is not weak.

Note that the definition of weakness is more relaxed than that of [10, 2], while still allowing us to use a linear-time symbolic model checking algorithm.

We can order SCCs according to their strength: Strong SCCs are stronger than weak ones, and weak SCCs that are not terminal are stronger than terminal SCCs. In general, the weaker an SCC is, the easier it is to decide language emptiness: An automaton containing a weak (terminal) SCC S has a nonempty language if $\mathsf{EF}\,\mathsf{EG}\,S \cap Q_0 \neq \emptyset$ ($\mathsf{EF}\,S \cap Q_0 \neq \emptyset$) holds.[1] The strength of an SCC-closed set or of an automaton is the maximum strength of its accepting SCCs.

3 Don't Care Conditions

Image and preimage usually account for most of the CPU time in symbolic, BDD-based model checking [4, 13]. Therefore, it is important to minimize the sizes of the representations of both the transition relation, and the argument to the (pre-)image computation. The size of a BDD is not directly related to the size of the set it represents. If we need not represent a set exactly, but can instead determine an interval in which it may lie, we can use known techniques [6, 16] to find a set within this interval with a small BDD representation.

Often, we are only interested in the results as far as they lie within a *care set* K (or outside a *don't care set* $\overline{K}$). Since the satisfaction of a property is only relevant within the set of reachable states R, we can use R as a care set to add or delete edges that emanate from unreachable states. By doing this, the image of a set that is contained within R remains the same. Likewise, the part of the preimage of a set S that intersects R remains the same, even if we add unreachable states to S. This use of R as care set depends on the fact that no edges from reachable to unreachable states are added.

The algorithm of Section 4 manipulates small portions of the state space, defined by SCC-closed sets. This allows us to use care sets that are often much smaller than the set of reachable states, and thus to increase the chance of finding small BDDs. We cannot use the approach outlined for the reachable states, since there may be edges from an SCC-closed set to other states. We show here that in order to use arbitrary sets as care sets in image computation, a 'safety zone' consisting of the preimage of the care set needs to be kept; similarly for preimage computation.

Theorem 1. *Let Q be a set of states and let $T \subseteq Q \times Q$ be a transition relation. Let $K \subseteq Q$ be a care set, $B \subseteq K$ a set of states,*

$$T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K}), \text{ and}$$
$$B \subseteq B' \subseteq B \cup \overline{\mathrm{pre}_{T'}(K)} \ .$$

Then, $\mathrm{img}_{T'}(B') \cap K = \mathrm{img}_T(B) \cap K$.

Proof. First, suppose that $q' \in \mathrm{img}_{T'}(B') \cap K$, and let $q \in B'$ be such that $q' \in \mathrm{img}_{T'}(\{q\}) \cap K$. Since $q' \in K$, we have $q \in \mathrm{pre}_{T'}(K)$. Hence, $q \in B'$ implies $q \in B$, and $q, q' \in K$, which means that $q' \in \mathrm{img}_T(\{q\}) \cap K$. Finally, $q \in B$ implies $q' \in \mathrm{img}_T(B) \cap K$. Conversely, suppose that $q' \in \mathrm{img}_T(B) \cap K$, and let $q \in B$ be such that $q' \in \mathrm{img}_T(\{q\}) \cap K$. Now $q, q' \in K$, and hence $q' \in \mathrm{img}_{T'}(\{q\}) \cap K$, and since $q \in B'$, $q' \in \mathrm{img}_{T'}(B') \cap K$. $\square$

[1] $\mathsf{EF}\,S$ is the subset of Q from which S is reachable, while $\mathsf{EG}\,S$ is the set of states in Q that are the origins of infinite paths entirely contained in S.

Hence, we can choose T' and B' to be sets within the given interval that have a small representation,[2] and use them instead of T and B. Through symmetry, we can prove the following theorem.

Theorem 2. *Let Q be a set of states and let $T \subseteq Q \times Q$. Let $K \subseteq Q$, $B \subseteq K$, $T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K})$, and $B \subseteq B' \subseteq B \cup \overline{\mathrm{img}_{T'}(K)}$. Then, $\mathrm{pre}_{T'}(B') \cap K = \mathrm{pre}_T(B) \cap K$.*

Edges are added to and from the set $\overline{K}$ (outside K) and the safety zone for (pre-)image computation excludes the immediate (successors) predecessors of K. Note that the validity of the aforementioned use of the reachable set as care set follows as a corollary of these two theorems.

4 SCC Refinement

This paper describes a hybrid algorithm for fair cycle detection that combines SCC refinement with more classical algorithms—like the one of Emerson and Lei [7]—that compute an SCC hull [15]. We shall here describe the general framework, and in later sections we shall discuss the implementation choices that we have made. The refinement processes uses a set of overapproximations of the system. We separate the generation of the overapproximations from their use; the method presented here works for any set of overapproximations.

4.1 Refinement

The core of our refinement approach is expressed by the following theorem.

Theorem 3. *Let $\mathcal{A}$ be a Büchi automaton, and let $\mathcal{A}_1, \ldots, \mathcal{A}_n \geq \mathcal{A}$ be overapproximations. Then, $\mathrm{SCCs}(\mathcal{A})$ is a refinement of*

$$\{C_1 \cap \cdots \cap C_n \mid C_1 \in \mathrm{SCCs}(\mathcal{A}_1), \ldots, C_n \in \mathrm{SCCs}(\mathcal{A}_n)\} \setminus \emptyset \ .$$

In particular, the set of SCCs of $\mathcal{A}$ is a refinement of the set of SCCs of any overapproximation $\mathcal{A}'$ of $\mathcal{A}$. Hence, an SCC of $\mathcal{A}'$ is an SCC-closed set (but not necessarily an SCC) of $\mathcal{A}$. This theorem allows us to gradually refine the set of SCCs until we arrive at the SCCs of $\mathcal{A}$.

One of the benefits of this approach is that we can often decide early that an SCC-closed set does not contain an accepting cycle. This allows us to trim the state space before considering the exact system, keeping around only ‘suspect’ SCCs.

Observation 4 *Let C be an SCC-closed set of $\mathcal{A}$. If $C \cap F_i = \emptyset$ for any $F_i \in \mathcal{F}$, then C has no states in common with any accepting cycle.*

We also have the following strength-reduction theorem, which allows us to use special algorithms for weak and terminal automata without analyzing the complete system.

[2] The actual use of don’t care information for the transition relation is asymmetric: We add edges out of don’t care states, but not into them. This is due to the useful minimizations that can be performed on a partitioned transition relation.

Theorem 5. *Let $\mathcal{A}$ and $\mathcal{A}'$ be Büchi automata with $\mathcal{A} \leq \mathcal{A}'$, and let $\mathcal{A}$ be complete. If C is a weak (terminal) set of $\mathcal{A}'$, then C is a weak (terminal) set of $\mathcal{A}$.*

The strength of a strong SCC-closed set may actually reduce in going from $\mathcal{A}'$ to $\mathcal{A}$. For example, a strong SCC may split into two weak ones.

4.2 Algorithm

The results of Section 4.1 motivate the algorithm GENERIC-REFINEMENT of Fig. 1. The algorithm takes as arguments a Büchi automaton $\mathcal{A}$ and a set L of overapproximations to $\mathcal{A}$, which includes $\mathcal{A}$ itself. The relation $\leq$ on L is not required to be a total order. The algorithm returns true if an accepting cycle exists in $\mathcal{A}$, and false otherwise.

The algorithm keeps a set Work of obligations, each consisting of a set of states, the series of approximations that have been applied to it, and an upper bound on its strength. Initially, the entire state space is in Work, and the algorithm keeps looping until Work is empty or a fair SCC has been found. The loop starts by selecting an element (S, L', s) from Work and a new approximation $\mathcal{A}'$ from L. If $\mathcal{A}' = \mathcal{A}$, the algorithm may decide to run a standard model checking procedure on the SCC at hand. Otherwise, it decomposes S into accepting SCCs, and after analyzing their strengths, adds them as new Work. The algorithm uses several subroutines.

Subroutine SCC-DECOMPOSE, takes an automaton $\mathcal{A}'$ and a set S, intersects the state space of $\mathcal{A}'$ with S to yield a new automaton $\mathcal{A}''$, and returns the set of accepting SCCs of $\mathcal{A}''$. Note that an SCC of $\mathcal{A}''$ is not necessarily an SCC of $\mathcal{A}'$. The subroutine discards any unfair SCCs, as justified by Observation 4. Subroutine ANALYZE-STRENGTH returns the strength of the set of states. (See Section 2.) Subroutine MODEL-CHECK (shown) returns true iff a fair cycle is found using the appropriate model-checking technique for the strength of the given automaton.

The way entries and approximations are picked is not specified, and neither is it stated when ENDGAME returns true. These functions can depend on factors such as the strength of the entry, the approximations that have been applied to it, and its order of insertion. In later sections we shall make these functions concrete.

It follows from Theorem 3 that at any point of the algorithm, for any entry (S, L', s) of Work, S is an SCC-closed set of $\mathcal{A}$. At any point, the sets of states in Work are disjoint. Termination is guaranteed by the finiteness of L and of the set of SCCs of $\mathcal{A}$.

When decomposing an SCC-closed set S, we can use the complement of S as a don't care condition as discussed in Section 3. Because S may be small in comparison to Q, this may lead to a significant improvement in efficiency.

The SCC-refinement algorithm computes much of the needed information about the SCCs of a system on overapproximations of it. Because these overapproximations are often much simpler than the concrete system, this approach may be very efficient. Because the SCCs of a system are a refinement of the SCCs of any overapproximation, any computation on an overapproximate system divides the state space into several components, some of which are thrown away without considering them in the exact system, and some of which are analyzed further in isolation.

At any point in time, we keep around an overapproximation of the reachable states to discard unreachable SCCs. Whenever we refine the system, we compute the set of

```
type Entry = record
    $S$;   // An SCC-closed set of $\mathcal{A}$
    $L'$;  // Set of approximations that have been considered
    $s$    // Upper bound on the strength of the SCC
end

MODEL-CHECK$(\mathcal{A}, S, s)$\{   // Automaton $\mathcal{A}$, SCC-closed set $S$, and its strength $s$
    case $s$ of
        strong: return $Q_0 \cap \mathsf{EG}_{\mathcal{F}}(S) \neq \emptyset$;   //call Emerson-Lei
        weak: return $Q_0 \cap \mathsf{EF}\,\mathsf{EG}(S) \neq \emptyset$;
        terminal: return $Q_0 \cap \mathsf{EF}(S) \neq \emptyset$
    esac
}

GENERIC-REFINEMENT$(\mathcal{A}, L)$\{   // Büchi automaton $\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$, and
                                          set of overapproximations $L$ with $\mathcal{A} \in L$
    var
        Work: set of Entry;

    Work = $\{(Q, \emptyset, \text{strong})\}$
    while Work $\neq \emptyset$ do
        Pick an entry $E = (S, L', s)$ from Work;
        Choose $\mathcal{A}' \in L$ such that there is no $\mathcal{A}'' \in L'$ with $\mathcal{A}'' \leq \mathcal{A}'$;

        if $\mathcal{A}' = \mathcal{A}$ and ENDGAME$(S, s)$ then
            if MODEL-CHECK$(\mathcal{A}, S, s)$ then
                return true
            fi
        else
            $\mathcal{C}$ := SCC-DECOMPOSE$(S, \mathcal{A}')$;
            if $\mathcal{C} \neq \emptyset$ and $\mathcal{A}' = \mathcal{A}$ then
                return true
            else
                for all $C \in \mathcal{C}$ do
                    $s$ := ANALYZE-STRENGTH$(C, \mathcal{A}')$;
                    insert $(C, L' \cup \{\mathcal{A}'\}, s)$ in Work
                od
            fi
        fi
    od
    return false
}
```

Fig. 1. The generic SCC-refinement algorithm

reachable states anew, limiting the search to the overapproximation that was obtained before. For this computation, we can use the overapproximation as care set. This scheme computes reachability multiple times, but [14] has shown that the use of approximate reachability information as a care set may more than compensate for that.

4.3 Underapproximations

The refinement approach that we have presented can be extended to the use of underapproximations. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be underapproximations of $\mathcal{A}$. First, as overapproximations can be used to discard the possibility of an accepting cycle, underapproximations can be used to assert their existence: if $\mathcal{A}_1$ contains an accepting cycle, then so does $\mathcal{A}$.

Furthermore, if an an SCC C_1 of $\mathcal{A}_1$ and an SCC C_2 of $\mathcal{A}_2$ overlap, then $\mathcal{A}$ contains an SCC $C \supseteq C_1 \cup C_2$. SCC-decomposition algorithms [18, 1] compute each SCC starting from a seed. The seed is usually a single state, but, in order to find C, we can use $C_1 \cup C_2$. Hence, we can avoid the recomputation of C_1 or C_2, and use $C_1 \cup C_2$ as a don't care set.

5 Composition

The SCC refinement algorithm described in Section 4 is generic because it does not specify: (1) What set of overapproximations L of the Büchi automaton $\mathcal{A}$ is available; (2) The rule to select the next approximation $\mathcal{A}'$ to be applied to a set S; (3) The priority function used to choose what element to retrieve from the Work set; and (4) The criterion used to decide when to switch to the endgame. These four aspects make up a *policy*; the first three are the subject of this section, while the fourth is discussed in Section 6.

5.1 Choice of the Approximations

We assume that $\mathcal{A}$ is the composition of a set of modules $M = \{M_1, \dots, M_n\}$, and that the set L of overapproximations consists of the compositions of subsets of M:

$$L \subseteq \{M_{i_1} \times \cdots \times M_{i_p} \mid \{i_1, \dots, i_p\} \subseteq \{1, \dots, n\}\} \ .$$

More flexible strategies (e.g., [9]) may generate larger sets of approximations and be compatible with our approach, but we shall not discuss them further.

We also assume that the states of $\mathcal{A}$ are the valuations of a set of r binary variables V; and that the sets of variables controlled by each module M_i are nonempty and form a partition of V. Hence, $n \leq r$ and for each $\mathcal{A}' \in L$ distinct from $\mathcal{A}$, $2\eta_{\mathcal{A}'} \leq \eta_{\mathcal{A}}$.

The set of all overapproximations generated from subsets of M forms a lattice, shown in Fig. 2 for $n = 4$. In the case illustrated by this figure, the coarsest approximation, which is the set of no modules, is the 1 of the lattice. (This approximation is never used in practice.) The exact system is the composition of all four modules. For sufficiently large n, it is impractical to make use of all 2^n overapproximations; consequently, we shall only consider efficient policies, in which a given state is contained in the set passed to SCC-DECOMPOSE $O(r)$ times.

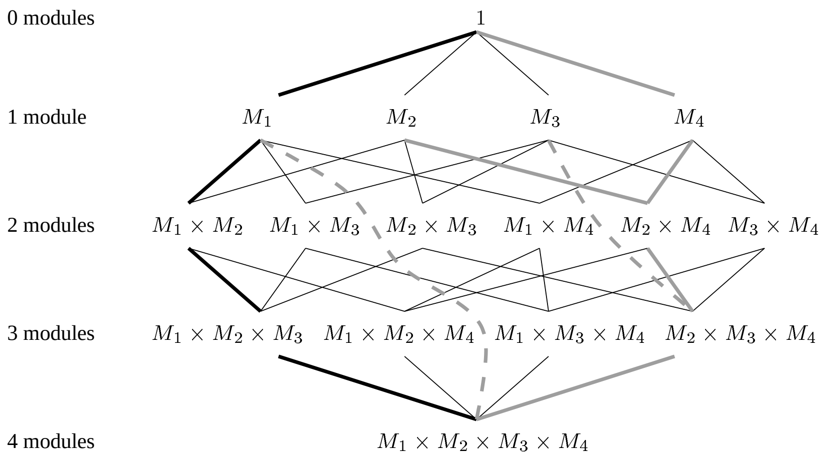

Fig. 2. Lattice of approximations

Specifically, we shall stipulate that there is a constant λ, such that L can be partitioned into subsets $L_1, \ldots, L_r$ satisfying the following conditions: (1) $|L_i| \leq \lambda$; (2) for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$; (3) $\mathcal{A} \in L_r$.

Two cases are illustrated in the figure. In both cases, $(j_1, \ldots, j_n)$ is a permutation of $(1, \ldots, n)$ that identifies a linear order of the modules. At the left of the figure (solid thick lines), the algorithm of Fig. 1 uses a *popcorn-line* policy with $(j_1, \ldots, j_4) = (1, 2, 3, 4)$ and $\lambda = 1$. The approximations are:

$$L = \{\mathcal{A}_i = M_{j_1} \times \cdots \times M_{j_i} \mid 1 \leq i \leq n\} \ .$$

When an entry $E = (S, L', s)$ is retrieved from Work, the $\mathcal{A}_i$ of lowest index that is not present in L' is chosen as the next approximation $\mathcal{A}'$.

At the right (thick grey lines), $2n - 1$ approximations are used in a *lightning-bolt* policy, for which $\lambda = 2$:

$$L = \{\mathcal{A}_{2i-1} = M_{j_1} \times \cdots \times M_{j_i} \mid 1 \leq i \leq n\} \cup \{\mathcal{A}_{2i} = M_{j_{1+1}} \mid 1 \leq i < n\} \ .$$

The selection of $\mathcal{A}'$ is done as in the previous case.

In Fig. 2, the order of the modules is $(4, 2, 3, 1)$. The approximations are:

$$\begin{array}{ll}
\mathcal{A}_1 = M_4, & \mathcal{A}_2 = M_2, \\
\mathcal{A}_3 = M_4 \times M_2, & \mathcal{A}_4 = M_3, \\
\mathcal{A}_5 = M_4 \times M_2 \times M_3, & \mathcal{A}_6 = M_1, \\
\mathcal{A}_7 = M_4 \times M_2 \times M_3 \times M_1. &
\end{array}$$

In both cases, the number of times a state is in the set passed to SCC-DECOMPOSE is bounded by the number of approximations in L. Therefore a popcorn-line policy tends to call SCC-DECOMPOSE fewer times, but a lightning-bolt policy may break up the SCC-closed sets with easy approximations ($\{\mathcal{A}_{2i}\}$) before applying to them harder approximations ($\{\mathcal{A}_{2i-1}\}$). Therefore, it tends to use less memory.

5.2 Complexity

The refinement algorithm described thus far cannot improve the worst-case complexity of the language emptiness check. Indeed, if all approximations distinct from $\mathcal{A}$ consist of one fair SCC, no benefit comes from the incremental SCC analysis. Under the stipulated conditions, however, it is easy to show that the incremental approach is within a constant factor from the non-incremental one.

Theorem 6. *If the set of approximations L can be partitioned into subsets $L_1, \ldots, L_r$ such that, for some constant λ, (1) $|L_i| \leq \lambda$; (2) for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$; and (3) $\mathcal{A} \in L_r$, then the generic refinement algorithm runs in $O(\eta_{\mathcal{A}} \log \eta_{\mathcal{A}})$ steps.*

Proof. The cost of SCC analysis for $\mathcal{A}'$ is bounded by $k\eta_{\mathcal{A}'} \log \eta_{\mathcal{A}'}$, for some constant k. Hence, the cost of analyzing all approximations and $\mathcal{A}$ itself is bounded by

$$k\eta_{\mathcal{A}} \log \eta_{\mathcal{A}} (\lambda + \lambda/2 + \lambda/4 + \cdots + \lambda/2^r) \enspace ,$$

which is bounded by $2\lambda k \eta_{\mathcal{A}} \log \eta_{\mathcal{A}}$. □

While we cannot hope for an improved run time in the worst case, we can expect that the refinement-based approach will be beneficial when the state space breaks up into many small SCC-closed sets. In particular, we can prove the following result.

Theorem 7. *Under the assumptions for L of Theorem 6, if for some constant γ, the pairs $(S, \mathcal{A}')$ passed to* SCC-DECOMPOSE *satisfy $|S| \leq \gamma \eta_{\mathcal{A}} / \eta_{\mathcal{A}'}$, then the refinement algorithm runs in $O(\eta_{\mathcal{A}})$ time.*

Proof. The analysis of $\mathcal{A}$ consists of the decomposition of SCC-closed sets of size bounded by γ. Their number is linear in $\eta_{\mathcal{A}}$, and each decomposition takes constant time. Hence, the total time for the analysis of $\mathcal{A}$ is $O(\eta_{\mathcal{A}})$. If $|C|$ is the number of states in SCC C of $\mathcal{A}'$, then $|C| \eta_{\mathcal{A}'} / \eta_{\mathcal{A}}$ is the *effective size* of C. The cost of analyzing $\mathcal{A}'$ is therefore $O(\eta_{\mathcal{A}'})$. With reasoning analogous to the one of Theorem 6, one finally shows that the total time is also $O(\eta_{\mathcal{A}})$. □

Another reason why the refinement-based approach may significantly outperform other algorithms is because it can discard large parts of the state space as soon as it establishes that they intersect no fair cycles by applying Observation 4. The advantage due to this ability to prune the search can be arbitrarily large.

5.3 Decomposition Trees

The popcorn-line approach, defines an SCC decomposition tree like the one of Fig. 3 that highlights the potential advantages of SCC refinement. The figure corresponds to a model of eight dining philosophers, with a property that states that under given fairness constraints, if a philosopher is hungry, she eventually eats. The system has nine modules. (The property automaton besides the philosophers.) The property passes, i.e., no fair cycles exist in the system. The tree of fair SCCs is shown. The nodes at Level i are the SCCs of $\mathcal{A}_i$. ($\mathcal{A}_1$ is the property automaton.) The size of each SCC is given; there

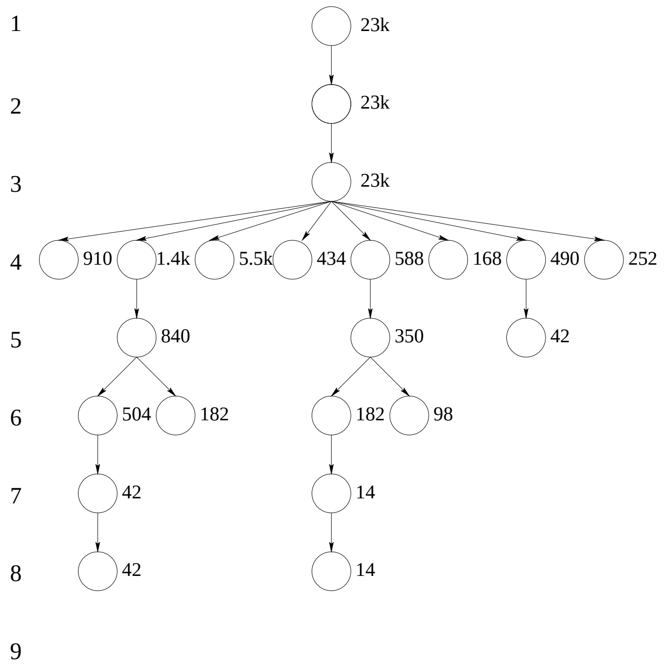

Fig. 3. SCC refinement tree

are about 47k reachable states. Note that only very small sets of states remain after the first four modules[3] are composed, and that very little work is done on the exact system. The effective size of each SCC is also shown in parentheses. Since the effective sizes correlate to the actual computational effort, the numbers of Fig. 3 show that the cost is quite modest at all levels of refinement.

To define a policy we need to specify the order in which elements are retrieved from the Work set. Two obvious choices are FIFO and LIFO order. As one would expect, the SCC refinement tree is traversed in breadth-first manner for a FIFO order, and in depth-first manner for a LIFO order. When, as in Fig. 3, there are no fair cycles in $\mathcal{A}$, the order in which the tree is visited is immaterial. However, in the presence of fair cycles, one strategy may lead to earlier termination than the other. If one assumes that fair cycles are numerous, then depth-first search is particularly attractive. Breadth-first search, on the other hand, can be implemented with low overhead.

6 Implementation and Experiments

We describe here details of two implemented policies for the SCC refinement algorithm D‘n’C, and of the experiments we ran. Both versions implement the popcorn-line ap-

[3] These four modules are the property automaton, the philosopher named in the property, and her two neighbors.

proach, with breadth-first search of the SCC refinement tree. Both heuristically partition the system to be verified according to the strategy of [9]. They then sort the modules according to their distances from the state variables of the property automaton.

The two policies differ in when they switch to the endgame: The first policy de-emphasizes compositionality in comparison to strength reduction by performing only two levels of composition. At the first level it computes the SCCs of the property automaton, and at the second level it composes all the other modules of the system.

The second policy tries to exploit the full compositionality implied by Figs. 2 and 3. For ease of reference, we refer to the first policy as the *Two-level* method, and to the second as the *Multi-Level* method.

In both policies, if any fair SCCs are present, the algorithm checks their strength. If any of them are weak, they are grouped together, and the exact system is checked for cycles within these SCCs. The underlying assumption is that model checking weak SCCs is much cheaper than model checking strong SCCs. If D‘n’C finds a cycle in the exact system, it terminates, otherwise it discards these SCCs. If no SCCs are present, the algorithm also terminates: there are no cycles. Otherwise, the approximate system is refined.

The Multi-Level method heuristically stops the refinement at some point, and then immediately composes all the remaining modules, thus proceeding directly to the exact system. Right now we are using a simple heuristic—we stop linear composition after 30% of the latches have been composed, and then “jump” to the the exact system to limit overhead, and to avoid having too many fair SCCs in the full SCC refinement tree. Once the exact system is reached, the Vis implementation of the Emerson-Lei algorithm is applied to each of its SCC-closed sets.

Our algorithm is implemented in Vis-1.4 [3], and the results of Table 1 were obtained by appropriately calling the standard Language Emptiness command of Vis. SCC analysis is performed with the Lockstep algorithm of [1] implemented as described in [15]. In Table 1, all examples were run with the same fixed order (obtained with dynamic variable reordering). For the same set of models and property automata, we also obtained a second table, with dynamic variable ordering turned on for each example. Similarly, we obtained a third table, using the EL2 variant of the Emerson-Lei algorithm [8]. Since the character of the results was not significantly different, the second and third tables were omitted for brevity.[4] All experiments were run on an IBM Intellistation running Linux with a 400MHz Pentium II processor with 1GB of SDRAM.

The table has four columns. The three fields of the first column give the name of the example, a symbol indicating whether the formula passes (P: no fair cycles exist) or fails (F: a fair cycle exists), and the number of binary latches in the system. The three fields of the second column, obtained by directly applying the Vis Emerson-Lei algorithm, give: (1) the time it took to run the experiment (Time/Out (T/O) indicates a run time greater than 14400s); (2) the peak number of live BDD nodes (in millions—the datasize limit was set to 750MB); and (3) the total number of preimage (EX) / image (EY) computations needed.

[4] The only exception to the statement was the fact that the example nmodem1 took only 209 seconds with EL2, versus 4384 for the original Emerson-Lei algorithm.

Table 1. Experimental results

			Emerson-Lei (Vis LE)			D'n'C Two-Level				D'n'C Multi-Level				
Circuit and LTL	P/ F	latch num	Time (s)	Bdd (M)	EX/EY	Time (s)	Bdd (M)	EX/EY	time ratio	Time (s)	Bdd (M)	EX/EY	time ratio	
bakery1	F	56	212	5.1	5337/0	31	1.3	354/4	14%	27	1.3	484/328	12%	
bakery2	P	49	69	3.4	526/0	20	1.3	10/4	28%	20	1.3	62/73	28%	n
bakery3	P	50	421	14	1593/0	46	2.5	90/4	10%	43	1.8	537/428	10%	
bakery4	F	58	T/O	-	-/-	1950	3.4	1088/5	<13%	1337	4.7	947/96	<9%	
bakery5	F	59	T/O	-	-/-	1009	6.1	127/5	<7%	623	6.1	216/243	<4%	
eisenb1	F	35	23	1.0	416/0	16	0.9	21/4	69%	16	0.9	21/4	69%	
eisenb2	F	35	T/O	-	-/-	4800	8.2	162/5	<33%	1683	7.7	105/93	<11%	w
elevator1	F	37	210	14	163/0	49	2.8	132/9	23%	41	2.2	155/31	19%	
nmodem1	P	56	4384	11	5427/0	192	1.1	992/4	4%	233	0.6	5007/71	5%	
peterson1	F	70	17	1.1	24/0	20	1.3	19/4	117%	21	1.2	157/173	123%	
philo1	F	133	371	12	258/0	7	0.2	8/12	1%	7	0.2	8/12	1%	w
philo2	F	133	73	2.8	557/0	30	1.3	258/5	41%	12	0.5	25/44	16%	w
philo3	P	133	T/O	-	-/-	T/O	-	-/-	-	115	1.2	993/224	<1%	
shamp1	F	143	44	2.1	8/0	103	5.6	9/6	234%	87	2.2	266/280	197%	
shamp2	F	144	T/O	-	-/-	1892	16.	74/6	<13%	101	2.9	345/349	<1%	
shamp3	F	145	T/O	-	-/-	337	4.4	19/17	<2%	335	4.4	19/17	<2%	w
twoq1	P	69	12	0.4	25/0	4	0.1	7/9	33%	4	0.1	7/9	33%	n
twoq2	P	69	241	8.9	175/0	27	0.8	91/5	11%	30	0.9	181/95	12%	

These same field descriptors also apply to the third and fourth columns (for the Two-Level and Multi-Level versions of the D'n'C algorithm), except that the latter has an additional field that indicates how the verification process terminates: 'n' means that the algorithm arrives at some intermediate level of the refinement process in which there no longer exists any fair SCC; 'w' means that there is a weak fair SCC found and it contains a fair cycle.

The property automata being used in the experiment are translated from LTL formulae. In order to avoid bias in favor of our approach, each model is checked against a *strong* LTL property automaton. Note that the presence of the n or w in the last field demonstrates that both pruning of the SCC refinement tree and strength reduction are active in these experiments.

We first compare the D'n'C algorithm to the one by Emerson and Lei. With only three exceptions out of 18 examples, the rows of the table indicate a significant (more than a factor of 2) performance advantage for the D'n'C algorithm.

Comparing the Two-Level and Multi-Level versions, one sees that on these examples, with four exceptions (eisenb2, philo2, philo3, and shamp2), the two policies give comparable performance. We think that this is because most of our examples are simple mutual-exclusion and arbitration protocols, in which the properties have little locality. We expect the compositional algorithm to do even better on models with more locality, and we are still enlarging the diversity of our sample set. On the other hand, one can see that the greater compositionality of the Multi-Level version proves its worth, especially on the larger examples.

7 Conclusions

In this paper we have presented a hybrid algorithm for fair cycle detection that uses abstractions to gradually refine the SCC-closed sets of a system to its SCCs. We have shown a general framework for SCC refinement and we have discussed different policies, based on the traversal of a lattice of overapproximate systems. Our algorithm has the advantages of being compositional, considering only parts of the complete state space, and taking into account the strength of an SCC to decide the proper model checking algorithm.

We have implemented two policies. In comparison to the original Emerson-Lei algorithm, our experimental results demonstrate significant and almost consistent performance improvement. This indicates the importance of all three improvement factors built into D‘n’C: (1) SCC refinement, (2) compositionality, and (3) strength reduction. Though the compositional approach does not improve the worst-case complexity over the algorithm of [1], we have identified conditions under which the proposed algorithm runs in linear time.

The D‘n’C algorithm can be highly parallelized by assigning different entries from the Work list to different processors. Processors that deal with disjoint sets of states have minimal communication and synchronization requirements. Although, the algorithm is geared towards symbolic model checking, SCC refinement can also be combined with explicit state enumeration approaches.

The experimental results show that even the simpler Two-Level policy performs very well in comparison to the Emerson-Lei algorithm. On all examples, the Multi-Level version of D‘n’C is either superior to, or comparable to the Two-Level version. We have noted that superiority occurs for the larger examples, and we speculate that D‘n’C will ultimately be able to handle some significantly larger examples. The simplicity of the implemented policies in comparison to the generality of Sections 4.1 and 5 suggests that there are many promising extensions and variations that so far remain experimentally unexplored. Among these, the joint application of over- and underapproximations is of special interest. Several iterative approaches to model checking have been proposed [11, 9, 5]. These approaches do not carry the SCC decomposition from one level of refinement to the next. On the other hand, they use counterexamples to guide refinement—something that is currently missing from our implementation, and that could improve its effectiveness.

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] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 154–169. Springer-Verlag, Berlin, July 2000.

[6] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, November 1990.

[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. 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.

[9] J.-Y. Jang. *Iterative Abstraction-based CTL Model Checking*. PhD thesis, University of Colorado, Department of Electrical and Computer Engineering, 1999.

[10] 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.

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

[12] 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.

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

[14] I.-H. Moon, J.-Y. Jang, G. D. Hachtel, F. Somenzi, C. Pixley, and J. Yuan. Approximate reachability don't cares for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 351–358, San Jose, CA, November 1998.

[15] 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.

[16] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proceedings of the Design Automation Conference*, pages 225–231, San Diego, CA, June 1994.

[17] 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.

[18] 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.