Compositional SCC Analysis for Language Emptiness[⋆]

Chao Wang[1], Roderick Bloem[2], Gary D. Hachtel[3], Kavita Ravi[4], and Fabio Somenzi[3]

[1] NEC Laboratories America, USA
[2] Graz University of Technology, Austria
[3] University of Colorado at Boulder, USA
[4] Cadence Design Systems, USA

Abstract. We propose a refinement approach to language emptiness, which is based on the enumeration and the successive refinements of SCCs on over-approximations of the exact system. Our algorithm is compositional: It performs as much computation as possible on the abstract systems, and prunes uninteresting part of the search space as early as possible. It decomposes the state space disjunctively so that each state subset can be checked in isolation to decide language emptiness for the given system. We prove that the strength of an SCC or a set of SCCs decreases monotonically with composition. This allows us to deploy the proper model checking algorithms according to the strength of the SCC at hand. We also propose to use the approximate distance of a fair cycle from the initial states to guide the search. Experimental studies on a set of LTL model checking problems prove the effectiveness of our method.

1 Introduction

1.1 Background and motivation

Checking language emptiness of a Büchi automaton is a core procedure in LTL [24, 36] and fair-CTL model checking [26] and in approaches to verification based on language-containment [21]. The cycle detection algorithms commonly used in symbolic model checkers fall into two categories [32]: One is based on the computation of an *SCC hull* [13, 17, 35, 19, 14, 34], and the other is based on SCC enumeration [42, 2, 3, 15]. Although some SCC enumeration algorithms [2, 3, 15] have better worst-case complexity bounds than the SCC hull algorithms—$O(\eta \log \eta)$ or $O(\eta)$ versus $O(\eta^2)$, where η is the number of states of the system—the comparative study of [32] shows that the

[⋆] Parts of this work appeared in preliminary form in [38] and [39].

worst-case theoretical advantage seldom translates in shorter CPU times. In many practical cases, applying any of these symbolic algorithms directly to the entire system to check language emptiness remains prohibitively expensive.

We regard the given model, or *exact system*, as the synchronous composition of many submodules; composing any subset of these submodules gives a simplified model, which may help in checking language emptiness for the original system. The removal of constraints from the excluded submodules makes the simplified model an over-approximation—it may have more behaviors than the exact system. As a result, verification of an LTL property with the simplified model may be conservative: If the language of the simplified model is empty, the language of the original system is also empty; if the language is not empty, however, the language of the original system may still be empty—that is, there may be a "false negative."

Although verification in an abstract model may result in a false negative, it does provide valuable information for the language emptiness checking of the original system. Given a model $\mathcal{A}$, every SCC in $\mathcal{A}'$, an over-approximation of $\mathcal{A}$, consists of one or more complete SCCs of $\mathcal{A}$. In other words, an SCC in the concrete system must be either included in or have no intersection with an SCC in the abstract model. Let π be the set of SCCs of $\mathcal{A}$; then, π is a *refinement* of the set of SCCs in $\mathcal{A}'$. In addition, if an SCC in the abstract model does not contain a fair cycle, none of its refinements will. Therefore, it is possible to enumerate the fair SCCs in $\mathcal{A}'$ first, and then refine each of them individually to compute the fair SCCs in $\mathcal{A}$.

We propose a compositional SCC analysis algorithm to language-emptiness checking, which is based on the enumeration and successive refinement of SCCs on a set of over-approximations of the exact system. By combining appropriate cycle-detection algorithms (SCC hull or SCC enumeration algorithms) into our general framework, we get a hybrid algorithm that shares the good theoretical characteristics of SCC enumeration algorithms, while outperforming the most popular SCC-hull algorithms, including the one of Emerson and Lei.

The analysis is conducted on a set of over-approximations of the exact system, ranging from the most abstract to the most concrete. Applying SCC enumeration on the most abstract over-approximation gives us the initial SCC partition of the state space. The partition is then refined on a more concrete over-approximation—one that is usually the composition of the previous over-approximation and a remaining submodule. At any stage, if an SCC does not contain any fair cycle, it is not considered in any more concrete systems. The procedure may refine parts of the state space until it reaches the exact system. However, each SCC of the exact system is contained in an SCC of each of its over-approximated abstract systems, and we do not have to consider further many non-fair SCCs. Hence, we can often drastically limit the set of states in which a fair cycle may be found.

In language emptiness checking, the models are considered as Büchi automata. The *strength* of a Büchi automaton [20, 4] is an important factor in checking the emptiness of its language. When $\mathcal{A}'$, an over-approximation of the exact system $\mathcal{A}$, is known to be *terminal* or *weak*, specialized algorithms exists for checking the emptiness of the language in $\mathcal{A}$. Previous work [4] showed that these specialized algorithms usually outperform the general language emptiness algorithms. However, the previous classification of *strong, weak* and *terminal* was applied to the entire Büchi automaton instead of each individual SCC. This can be inefficient, because a Büchi automaton with a strong SCC and several weak ones is classified as strong. In this paper, we apply the definition of strength to each individual SCC, so that the appropriate model checking procedure can be deployed at a finer granularity. Furthermore, we prove that the fair SCC strength of an automaton may not increase as more submodules are composed—which we call the strength-reduction theorem: After the composition, a strong SCC may break into several weak SCCs, but a weak one cannot generate strong SCCs. Our method analyzes SCCs as they are computed to take maximal advantage of their weakness.

Our approach achieves favorable worst-case complexity bound: $O(\eta)$ or $O(\eta \log \eta)$, depending on what underlying SCC enumeration algorithm is used; this is valid even when the algorithm adds one submodule at the time to the abstract system until the concrete system is reached. In practice, however, the effort spent on the abstract systems can be justified only if it does not incur too much overhead. As the abstract system becomes more and more concrete through composition, the SCC enumeration on the abstract system may become too expensive. In such cases, the algorithm jumps directly to the concrete system, with all the useful information gathered from the abstract systems.

Based on the SCC quotient graph of the last abstract model, we *disjunctively* decompose the concrete state space into subspaces. Each subspace induces a Büchi subautomaton that is an under-approximation of the exact system; therefore, it accepts a subset of the original language. The decomposition is *exact*, for the union of these language subsets is the original language. Therefore, language emptiness of the exact system can be checked in each of these subautomata in isolation. By focusing on one subspace at a time, we mitigate the BDD explosion during the most expensive part of the computation—testing abstract fair cycles against the concrete system.

To further speed up the search for fair cycles, we propose a guided search algorithm for the traversal of the subspaces. Note that we are only interested in the reachable state space. Early termination is promoted by examining first the promising areas where fair cycles may reside, and by stopping the cycle-detection algorithm as soon as a fair cycle is found. In the targeted search, the approximate distance to the fair SCCs is used as guidance.

To summarize, our compositional SCC analysis algorithm, called the *Divide and Compose (D‘n’C)* algorithm, has the following features:

- It is compositional and performs as much work as possible on abstracted systems.
- It considers only parts of the state space at any time.
- It uses the *strength* of a given set of SCCs to decide the proper model checking algorithm.
- It localizes the fair-cycle detection by disjunctive decomposition and targeted search.

1.2 Related work

The previous works most closely related to ours are the various abstraction refinement algorithms [21, 1, 22, 30, 18, 9, 41, 10, 7, 27, 23, 40, 16]. Abstraction refinement was first introduced by Kurshan [21] to check linear properties specified by ω-regular automata. For these universal properties, it is enough to consider abstract models that are over-approximations; they are obtained by considering subsets of the variables, while leaving the others unconstrained. (This definition of the abstract systems is essentially the same as ours.) If the verification result is inconclusive, the current abstract model is refined by adding more details of the system, after which the property is checked again.

However, in all these existing abstraction refinement methods, information gathered from previous abstract models is seldom used to help the verification on the current abstract model; model checking starts from scratch in every iteration. In [25], previously computed satisfying states are used as the starting point for the backward fix-point. The set of states, however, are under-approximations of the exact satisfying states. In contrast, we stress the importance of carrying information learned from previous over-approximated abstract systems to the next. We present a general framework that allows us to simplify a search using information obtained in a more abstract version of the state space. In particular, the searches in abstract state space allow us to forgo searching parts of the concrete state space, and even if search is required, this information may allow us to use cheaper algorithms.

On the other hand, the selection of abstract systems in our algorithm is very general, and our implementation is rather primitive compared to that of the aformentioned abstraction refinement algorithms, particularly the counter-example guided methods [9, 41, 10, 7, 40, 23]. These works have addressed the details on how the next abstract system should be selected, based on the analysis of the spurious abstract counter-example(s). Although the majority of them have been designed for checking safety properties, we believe that these refinement methods can be adapted to the language emptiness check, and therefore, can be harmoniously combined with our compositional SCC analysis algorithm.

1.3 Organization of this paper

The rest of this paper is organized as follows: After reviewing the technical background in Section 2, we give the set of theorems underlying our compositional SCC analysis algorithm in Section 3 and discuss don't care conditions in Section 4. We present the generic algorithmic framework in Section 5 and the various compositional approaches in Section 6. Section 7 deals with disjunctive state space decomposition and the guided search for fair cycles. Implementation details and our experiments are discussed in Section 9; the results show that our new algorithm often achieves substantial savings in memory and CPU time. Section 10 summarizes the contributions of the paper and outlines promising future work.

2 Preliminaries

We model the system to be verified as a *labeled, generalized* Büchi automaton, defined as follows:

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

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

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 a finite alphabet for which a set P of atomic propositions is given and $A = 2^P$, and $\Lambda : Q \to A$ is the labeling function.

Note that we have defined automata with labels on the states, not the edges. The automata are generalized because multiple acceptance conditions are possible. 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. In this paper, all automata are assumed to be complete.

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 $q \in F_i$ 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 $\mathcal{A}$. $\mathcal{L}(\mathcal{A})$ is nonempty if and only if $\mathcal{A}$ contains a reachable *fair cycle*—that is, a cycle that is reachable from some initial state and intersects all acceptance conditions.

A Strongly-Connected Component (SCC) is a maximal set of states such that there is a path between any two states of the set. An SCC that consists of just one state without a self-loop is called *trivial*. A non-trivial SCC that intersects all acceptance conditions is called a *accepting (fair) SCC*. An SCC that contains some initial states is called an *initial SCC*. Contracting every SCC into a single node, merging parallel edges, and then removing self-loops results in the *SCC (quotient) graph*. The SCC graph of $\mathcal{A}$, denoted by $\mathcal{G}(\mathcal{A})$, is a Directed Acyclic Graph (DAG); it induces a partial order: a minimal (maximal) SCC has no incoming (outgoing) edge.

An *SCC-closed set* is the union of a collection of SCCs. The set of SCCs of $\mathcal{A}$, denoted by $\pi(\mathcal{A})$, is a partition of Q. An SCC partition π_1 is a *refinement* of another partition π_2 if, for every $C_1 \in \pi_1$, there exists $C_2 \in \pi_2$ such that $C_1 \subseteq C_2$.

We define the exact system $\mathcal{A}$ as the synchronous composition of several submodules. Composing a subset of these submodules gives us an over-approximated abstract model $\mathcal{A}'$. In symbolic algorithms, $\mathcal{A}$, $\mathcal{A}'$, as well as the submodules, are all defined over the same state space and the same set of atomic propositions, 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.

Definition 2. *The composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ of two Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$, where*

$$\mathcal{A}_1 = \langle Q, Q_{01}, T_1, \mathcal{F}_1, A, \Lambda \rangle \text{ and}$$
$$\mathcal{A}_2 = \langle Q, Q_{02}, T_2, \mathcal{F}_2, A, \Lambda \rangle \ ,$$

is a Büchi automaton $\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$ such that $Q_0 = Q_{01} \cap Q_{02}$, $T = T_1 \cap T_2$, and $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$.

Let $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$; both $\mathcal{A}_1$ and $\mathcal{A}_2$ can be considered as over-approximations of $\mathcal{A}$. An SCC in $\mathcal{A}_1$ or $\mathcal{A}_2$ is a SCC-closed set in $\mathcal{A}$.

Definition 3. *The* strength *of an accepting SCC C is defined as follows (cf. [20, 4]):*

- *C is* weak *if all cycles contained within it are accepting.*
- *C is* terminal *if it is weak and for every state $q \in C$, all the successor states of q are in some terminal SCCs. Terminality implies acceptance of all runs reaching C.*
- *C is* strong *if it is not weak.*

Note that strength is defined only for accepting SCCs. The strength of an SCC-closed set containing at least one accepting SCC or of an automaton is the maximum strength of its accepting SCCs. Our definition of weakness is more relaxed than that of [20, 4], while still allowing us to use faster symbolic model checking algorithms.

Lemma 1. *Given a Büchi automaton $\mathcal{A}$ and an overapproximation $\mathcal{A}'$ of $\mathcal{A}$,*

1. *if C is a weak SCC of $\mathcal{A}'$, then $\mathcal{A}$ contains a reachable fair cycle contained inside C if and only if $\mathcal{A} \models \mathsf{EF}\,\mathsf{EG}\, C \cap Q_0 \neq \emptyset$;*
2. *if C is a terminal SCC of $\mathcal{A}'$, then in $\mathcal{A}$ there is a reachable fair cycle contained in C if and only if $\mathcal{A} \models \mathsf{EF}\, C \cap Q_0 \neq \emptyset$.*

Proof. $\mathsf{EF}\, C$ is the subset of states in Q that can reach the states in C, while $\mathsf{EG}\, C$ is the subset of states in C that lead to a cycle lying in C. Assuming the definitions of weak and terminal, we have the proof as follows:

1. Suppose that C is a weak SCC of $\mathcal{A}'$, and $\mathcal{A}$ contains a reachable fair cycle inside C. Then $A \models \mathsf{EF}\,\mathsf{EG}\, C \cap Q_0 \neq \emptyset$. Conversely, if $A \models \mathsf{EF}\,\mathsf{EG}\, C \cap Q_0 \neq \emptyset$, then $\mathcal{A}$ contains a reachable cycle inside C. Since every cycle in $\mathcal{A}$ is also contained in $\mathcal{A}'$ and every cycle that $\mathcal{A}'$ contains in C is accepting, every cycle that $\mathcal{A}$ contains in C is accepting. Therefore $\mathcal{A}$ contains a reachable fair cycle.
2. Suppose that C is a terminal SCC of $\mathcal{A}'$, and $\mathcal{A}$ contains a reachable fair cycle inside C. Then a state of C is reachable in $\mathcal{A}$ and $A \models \mathsf{EF}\, C \cap Q_0 \neq \emptyset$. Conversely, suppose that $\mathcal{A} \models \mathsf{EF}\, C \cap Q_0 \neq \emptyset$. Note that all cycles in $\mathcal{A}'$ contained in C are accepting, and that there are no other infinite paths in $\mathcal{A}'$ that start in C. Hence, all cycles in $\mathcal{A}$ contained in C are accepting, and that there are no other infinite paths in $\mathcal{A}$ that start in C. Since one state of C is reachable and the transition relation is complete, a fair path within C is reachable in $\mathcal{A}$.

In symbolic algorithms, both the sets of states and the transition relation T are represented by their characteristic functions, which in turn are represented by Binary Decision Diagrams (BDDs [6]). These BDDs are manipulated as sets by union, intersection, complementation, as well as image EY and preimage EX computations. The image (or preimage) of a set of states consists of all their successors (or predecessor) in the graph. They are defined as follows:

$$\begin{aligned} \mathsf{EY}_T(S) &= \{q' \mid \exists q \in S : (q, q') \in T\} \\ \mathsf{EX}_T(S) &= \{q \mid \exists q' \in S : (q, q') \in T\} \ . \end{aligned}$$

When the context allows it, we shall drop the subscript identifying the transition relation.

Many interesting properties can be evaluated by fix-point computations, in which EX and EY are the basic symbolic steps. The set of states reachable from the initial states Q_0, denoted by $\mathsf{EP}\, I$, is a least fix-point computation

$$\mu Z \,.\, Q_0 \cup \mathsf{EY}(Z) \ .$$

Likewise, $\mathsf{EF}\, C$ is

$$\mu Z \,.\, C \cup \mathsf{EX}(Z) \ .$$

The computations of EG, EU, and $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$ are defined as follows:

$$\begin{aligned} \mathsf{EG}\, S &= \nu Z \,.\, S \cap \mathsf{EX}\, Z \\ \mathsf{E}\, S_1 \,\mathsf{U}\, S_2 &= \mu Z \,.\, S_2 \cup (S_1 \cap \mathsf{EX}\, Z) \\ \mathsf{EG}_{\mathsf{fair}}\,\mathsf{true} &= \nu Z \,.\, \mathsf{EX} \bigcap_{f_i \in \mathcal{F}} \mathsf{E}\, Z \,\mathsf{U} (Z \cap f_i) \ . \end{aligned}$$

In general, language emptiness can be decided by evaluating $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$. The computation of $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$ as described above is known as the Emerson and Lei algorithm [13], a representive of the SCC hull algorithms. It requires $O(\eta^2)$ preimage computations in the worst case, where η is the number of states.

An alternative way of checking for language emptiness is to enumerate all SCCs, and then check if they satisfy all the acceptance conditions. An SCC enumeration algorithm often picks a state v as a *seed*, and then computes both $\mathsf{EF}\, v$ and $\mathsf{EP}\, v$. The intersection of $\mathsf{EF}\, v$ and $\mathsf{EP}\, v$ gives the SCC containing v. After removing this SCC, the algorithm picks another seed among the remaining states. This procedure terminates when no state is left. Among the existing SCC enumeration algorithms, the one by Gentilini et al. [15] has the best complexity bound, which is $O(\eta)$. Note that it is better than the quadratic bound of SCC hull algorithms.

The fact that all components are defined over the same state space masks the fact that the analysis of a component is typically much easier than the analysis of the exact system. Complexity bounds given in terms of the numbers of states do not recognize this difference. In the following, we define the *effective number of states* to take into consideration the fact that an automaton does not control some of the state variables. Let V be a finite set of binary state variables, whose valuations form the set of states Q; then, $\eta = 2^{|V|}$. 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 $(q_1, q_2) \in T$ but $(q_1, q_2^v) \notin T$. In other words, v is controlled by $\mathcal{A}$ if the automaton can change its value. Let $V_{\mathcal{A}}$ be the subset of variables controlled by $\mathcal{A}$, the effective number of states of $\mathcal{A}$ is defined as $\eta_{\mathcal{A}} = 2^{|V_{\mathcal{A}}|}$. The language emptiness for $\mathcal{A}$ can be checked in $O(\eta_{\mathcal{A}}^2)$ steps by the Emerson-Lei algorithm, and in $O(\eta_{\mathcal{A}})$ steps by the algorithm of [15].

Image and preimage computations usually account for most of the CPU time in BDD-based symbolic model checking. 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 set of states. 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 generalized cofactors [11, 12] 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 language emptiness problem is only concerned with the set of reachable states R, we can regard R as a care set, and 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 unreachable states are introduced to S by adding edges. This use of the states in $\overline{R}$ as don't cares, which are often called the Reachability Don't Cares (RDCs), depends on the fact that no edges from reachable to unreachable states are added.

3 SCC refinement

We start with the definition of over-approximations of a Büchi automaton, followed by the theorems that provide the foundations of our SCC refinement algorithm. Automaton $\mathcal{A}'$ is an *over-approximation* of $\mathcal{A}$, denoted by $\mathcal{A} \leq \mathcal{A}'$, if $Q = Q'$, $Q_0 \subseteq Q_0'$, $T \subseteq T'$, $\mathcal{F} = \mathcal{F}'$, and $\Lambda = \Lambda'$. The relation $\leq$ is a partial order on automata; furthermore, $\mathcal{A} \leq \mathcal{A}'$ implies that $\mathcal{A}'$ simulates $\mathcal{A}$ [28]; hence, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.

Theorem 1 (Compositional refinement). *Let $\mathcal{A}, \mathcal{A}_1, \dots, \mathcal{A}_n$ be Büchi automata such that $\mathcal{A} \leq \mathcal{A}_i$ for $1 \leq i \leq n$. Then, the set of SCCs $\pi(\mathcal{A})$ is a refinement of*

$$\Theta = \{C_1 \cap \cdots \cap C_n \mid C_i \in \pi(\mathcal{A}_i)\} \setminus \emptyset \enspace .$$

Proof. Every state in an SCC $C \in \pi(\mathcal{A})$ is reachable from all other states in C. An over-approximation $\mathcal{A}_i$ preserves all transitions of $\mathcal{A}$, which means that in $\mathcal{A}_i$, every state in C remains reachable from the other states in C. Therefore, for $1 \leq i \leq n$, C is contained in an SCC of $\mathcal{A}_i$; hence it is contained in their intersection, which is an element of Θ. Since the union of all SCCs of $\mathcal{A}$ equals Q and distinct elements of Θ are disjoint, Θ is a partition of Q, and $\pi(\mathcal{A})$ is a refinement of it. □

In particular, $\pi(\mathcal{A})$ is a refinement of the SCC partitions of any of its over-approximations; thus, an SCC of $\mathcal{A}' \geq \mathcal{A}$ is an SCC-closed set of $\mathcal{A}$. Theorem 1 allows us to gradually refine the set of SCCs on the over-approximations until we arrive at $\pi(\mathcal{A})$. One benefit is that we can often decide early that an SCC-closed set does not contain an accepting cycle.

Observation 1 *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.*

Therefore, we can trim the state space by avoiding refining non-fair SCC-closed sets as soon as possible, keeping around only "suspect" SCCs.

Theorem 2 (Strength reduction). *Let $\mathcal{A}$ and $\mathcal{A}'$ be Büchi automata such that $\mathcal{A}$ is complete and $\mathcal{A} \leq \mathcal{A}'$. (Hence, $\mathcal{A}'$ is also complete.) If C is a weak (terminal) SCC-closed set of $\mathcal{A}'$, and it contains an accepting cycle of $\mathcal{A}$, then C is a weak (terminal) SCC-closed set of $\mathcal{A}$.*

Proof. We prove this by contradiction. Assume that C is a weak set of $\mathcal{A}'$, but is a strong set of $\mathcal{A}$. Then, at least one cycle in C is not accepting in $\mathcal{A}$. As an over-approximation,

$\mathcal{A}'$ preserves all paths of $\mathcal{A}$, including this non-accepting cycle, which makes C a strong set of $\mathcal{A}'$ too. However, this contradicts the assumption that C is weak in $\mathcal{A}'$. Therefore, C cannot be a strong set of $\mathcal{A}$. A similar argument applies to the terminal case. □

In other words, the strength of an SCC-closed set never increases as a result of composition. In fact, the strength may actually reduce in going from $\mathcal{A}'$ to $\mathcal{A}$. For example, a strong SCC may be refined into one or more SCC none of which is strong; a weak SCC may be refined into one or more SCCs, none of which is weak. This strength reduction theorem allows us to use special algorithms as soon as a strong SCC-closed set becomes weak or terminal.

Deciding the strength of an SCC-closed set can be expensive. In the implementation, we make conservative decisions of the strength of an SCC C as follows:

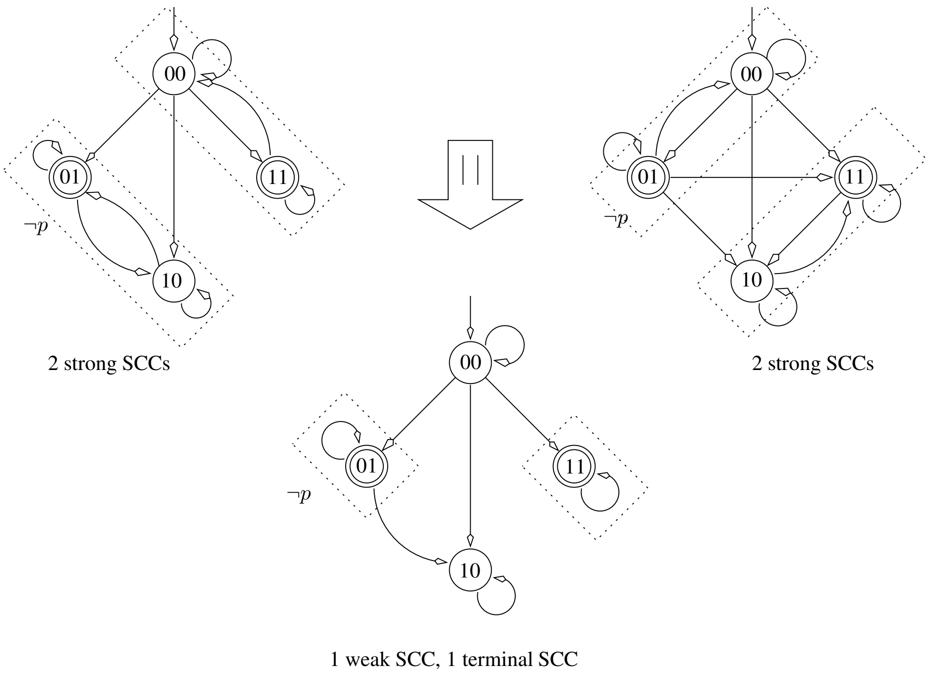
- C is weak if $C \subseteq F_i$ for every $F_i \in \mathcal{F}$;
- C is terminal if C is weak and $(\mathsf{EY}\, C) \setminus C = \emptyset$;
- C is strong otherwise.

We use the example in Fig. 1 to show the impact of composition. The three Büchi automata with one acceptance condition ($\mathcal{F} = \{F_1\}$) are defined on the same state space. State 01 is labeled $\neg p$; all other states are labeled $\mathtt{true}$ implicitly. Double circles indicate that the state satisfies the acceptance condition. The synchronous composition of the two automata at the top produces the automaton at the bottom. Note that only transitions that are allowed by both parent automata appear in the composed system. Both automata at the top are strong, although their SCC partitions are different. The composed system, however, has a weak SCC, a terminal SCC, and two non-fair SCCs. Its SCC partition is a refinement of both previous partitions.

Let the SCC quotient graph $\mathcal{G}(\mathcal{A})$ of a Büchi automaton $\mathcal{A}$ be $\mathcal{G} = \langle \mathcal{C}, \mathcal{C}_0, T_C, F_C \rangle$, where $\mathcal{C}$ is the set of SCCs, $\mathcal{C}_0 \subseteq \mathcal{C}$ is the set of initial SCCs, $T_C \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation, and F_C is the set of fair SCCs. Let $\mathcal{G}'$ be a subgraph of $\mathcal{G}$, where $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{C}'_0 \subseteq \mathcal{C}_0$, $T'_C \subseteq T_C$, and $F'_C \subseteq F_C$. In other words, removing some nodes or transitions, or making some fair nodes non-fair, gives us a subgraph. A subgraph of $\mathcal{G}(\mathcal{A})$ induces a subautomaton.

Definition 4. *Given the SCC quotient graph $\mathcal{G}$ of $\mathcal{A}$ and a subgraph $\mathcal{G}' = \langle \mathcal{C}', \mathcal{C}'_0, T'_C, F'_C \rangle$, the subautomaton $\mathcal{A} \Downarrow \mathcal{G}' = \langle Q, Q'_0, T', \mathcal{F}', A, \Lambda \rangle$ is defined as follows:*

- *$Q'_0 \subseteq Q_0$ is the subset of initial states that appear in $\mathcal{C}'_0$,*
- *$T' \subseteq T$ is the subset of transitions among the states that appear in $\mathcal{C}'$.*



2 strong SCCs

2 strong SCCs

1 weak SCC, 1 terminal SCC

Fig. 1. The impact of composition on the SCCs. Dash boxes represent fair SCCs

– *$F'_i \in \mathcal{F}'$ is the subset of $F_i \in \mathcal{F}$ that intersects the set of states in F'_C.*

Essentially, $\mathcal{A} \Downarrow \mathcal{G}'$ is the original automaton restricting its operation only in the set of states Q'. Therefore, $\mathcal{A} \Downarrow \mathcal{G}(\mathcal{A}) = \mathcal{A}$. Since accepting runs in the subautomaton are always accepting in $\mathcal{A}$, its language is a subset of $\mathcal{L}(\mathcal{A})$. Furthermore, the *pruning* operation on the SCC graph, defined as removing nodes that are not on any path from initial nodes to fair nodes, does not change the language accepted by the corresponding automaton. The claim can be extended to the SCC subgraph of any over-approximation of $\mathcal{A}$.

Observation 2 *Let $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{G}'$ be a subgraph of $\mathcal{G}(\mathcal{A}')$, then $\mathcal{L}(\mathcal{A} \Downarrow \mathcal{G}') \subseteq \mathcal{L}(\mathcal{A})$.*

We define an SCC subgraph $\mathcal{G}^{C_j}(\mathcal{A})$ for every fair node C_j of $\mathcal{G}(\mathcal{A})$, so that it contains all SCCs that are on the paths from the initial SCCs to C_j (including C_j); furthermore, all the nodes are marked non-fair, except for C_j. It can be constructed by marking C_j fair and all the other nodes non-fair and then pruning the SCC graph. When the context is clear, we will simply use $\mathcal{G}^{C_j}$ to denote such a subgraph. Recall that a Büchi automaton has an accepting run if and only if some reachable fair nodes exist in its SCC graph. It follows that each of these SCC subgraphs induces a subautomaton that accepts a subset of the original language; the union of these subsets of languages is the same as the language of the original automaton.

In addition, $\mathcal{G}^{C_j}$ can be further decomposed into subgraphs. An SCC subgraph of this kind, denoted by $\mathcal{G}_i^{C_j}$, represents a path from the i-th initial SCC to the j-th fair SCC. Therefore, the languages accepted by the subautomata $\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}$ also form a disjunctive decomposition of the language accepted by $\mathcal{A} \Downarrow \mathcal{G}^{C_j}$. The claim can be extended to the SCC subgraphs of any over-approximation of $\mathcal{A}$. To summarize, we have the following theorem:

Theorem 3 (Disjunctive decomposition). *Let $\mathcal{A} \leq \mathcal{A}'$ and the set of SCC subgraphs $\{\mathcal{G}_i^{C_j}\}$ be the disjunctive decomposition of $\mathcal{A}'$. Then, $\mathcal{L}(\mathcal{A}) = \emptyset$ if and only if $\mathcal{L}(\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}) = \emptyset$ for every subgraph.*

Subautomata with respect to the SCC subgraphs are *under-approximations* of the exact system. An under-approximation normally can be used to certify the existence of fair runs, but not to prove language emptiness. However, by Theorem 3 our disjunctive decomposition creates a complete set of under-approximations, therefore does not produce conservative results for language emptiness checking. An advantage of applying this disjunctive decomposition theorem is to decide language emptiness of the exact

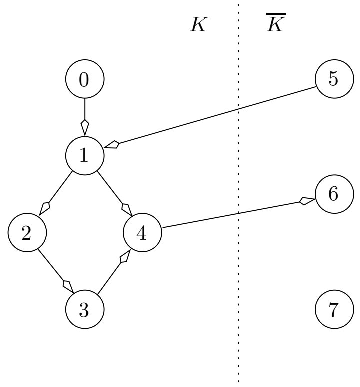


Fig. 2. Use of don't cares in the computation of SCCs

system by checking subautomata separately. When we restrict the search to a smaller state space, we increase the effectiveness of don't cares in speeding up symbolic image and preimage computations.

4 Don't care conditions

Thanks to the theorems in the previous section, at any time, we only manipulate small portions of the state space, defined by SCC-closed sets or subautomata. 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 (in Section 2) directly, since there may be edges from an SCC-closed set to other states as the one from State 4 to State 6 in Fig. 2. 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, the safety zone must consist of the image of the care set.

Theorem 4. *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, $B \subseteq K$ be a set of states. Finally, let $T' \subseteq Q \times Q$ be a transition relation and $B' \subseteq Q$ a set of states such that*

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

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

Proof. First, suppose that $q' \in \mathsf{EY}_{T'}(B') \cap K$, and let $q \in B'$ be such that $q' \in \mathsf{EY}_{T'}(\{q\}) \cap K$. Since $q' \in \mathsf{EY}_{T'}(\{q\})$, so $q \in \mathsf{EX}_{T'}(q')$, and because $q' \in K$, we have $q \in \mathsf{EX}_{T'}(K)$. Hence, $q \in B'$ implies $q \in B$, and $q, q' \in K$, which means that $q' \in \mathsf{EY}_T(\{q\}) \cap K$. Finally, $q \in B$ implies $q' \in \mathsf{EY}_T(B) \cap K$.

Conversely, suppose that $q' \in \mathsf{EY}_T(B) \cap K$, and let $q \in B$ be such that $q' \in \mathsf{EY}_T(\{q\}) \cap K$. Now $q, q' \in K$, and hence $q' \in \mathsf{EY}_{T'}(\{q\}) \cap K$, and since $q \in B'$, $q' \in \mathsf{EY}_{T'}(B') \cap K$. □

Hence, we can choose T' and B' within the given intervals so that they have a small representations, and use them instead of T and B. Through symmetry, we can prove the following theorem.

Theorem 5. *Let Q be a set of states and let $T \subseteq Q \times Q$. Let $K \subseteq Q$, $B \subseteq K$, $T' \subseteq Q \times Q$, and $B' \subseteq Q$ be such that*

$$T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K}), \textit{ and}$$
$$B \subseteq B' \subseteq B \cup \overline{\mathsf{EY}_{T'}(K)} \ .$$

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

Edges are added to and from states in the set $\overline{K}$ (states outside K), while 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 states as care set follows as a corollary of these two theorems. Figure 3 shows a possible choice of T' given the T and K of Fig. 2. For that choice of T', it shows, enclosed in the dotted line, the set $\overline{\mathsf{EX}_{T'}(K)}$. If $B = \{1, 2\}$, then $\mathsf{EY}_T(B) \cap K = \{2, 3, 4\}$. Suppose $B' = \{1, 2, 4\}$. Then

$$\mathsf{EY}_{T'}(B') \cap K = \{2, 3, 4, 6\} \cap \{0, 1, 2, 3, 4\} = \{2, 3, 4\} = \mathsf{EY}_T(B) \cap K \ .$$

Note that the addition of the edge from State 7 to State 3 causes the former to be excluded from $\overline{\mathsf{EX}_{T'}(K)}$.

5 The generic algorithm

The theorems of Section 3 motivate the generic SCC refinement algorithm in Fig. 4. The Divide and Compose (D‘n’C) algorithm, whose entry function is GENERIC-REFINEMENT, takes as arguments a Büchi automaton $\mathcal{A}$ and a set L of over-approximations to $\mathcal{A}$,

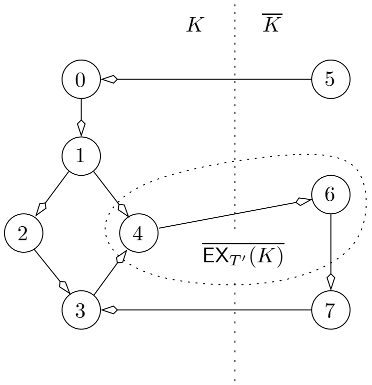


Fig. 3. Example of application of Theorem 4

which includes $\mathcal{A}$ itself. The relation $\leq$ on L is not required to be a total order. The algorithm returns true if a fair 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 over-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 to 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 $\mathcal{A}'$: It may be a proper subset of one. The subroutine avoids working on any non-fair SCCs, as justified by Observation 1. Subroutine ANALYZE-STRENGTH returns the strength of the set of states. (See Definition 3.) Subroutine MODEL-CHECK returns true if and only if a fair cycle is found using the appropriate model-checking technique for the strength of the given SCC.

The way entries and over-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 over-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 1 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 $\overline{S}$ as don't care set as discussed in Section 4. This usually gives us a much larger don't care set than the reachability don't cares; therefore, the use of don't cares can lead to a significant improvement in the computation efficiency. Furthermore, image and preimage computations on the over-approximations can be very cheap, because these over-approximations are usually much smaller than the concrete system. In addition, each SCC-closed set is divided into several components, some of which (the non-fair ones) are not considered in the exact system, and some of which are analyzed further in isolation.

Since we are not interested in unreachable states, we keep track of the reachable states of the current over-approximation to discard unreachable SCCs. Whenever the next subsystem is picked, we compute the set of reachable states anew, but limiting this computation to the previous reachable states; the new reachable states are contained in the previous reachable states, as long as the new abstract system is a refinement of the previous one. Because of this, previous reachable states can be used as a care set in computing the new ones. Although we compute reachability multiple times, [29] has shown that the use of approximate reachability information as a care set may more than compensate for the overhead.

The refinement approach that we have presented can be extended to the use of under-approximations. As over-approximations can be used to discard the possibility of an accepting cycle, under-approximations can be used to assert its existence. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be under-approximations of $\mathcal{A}$; if either $\mathcal{A}_1$ or $\mathcal{A}_2$ contains an accepting cycle, then so does $\mathcal{A}$. Furthermore, if 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-enumeration algorithms [42, 2, 3, 15] compute each SCC from a seed state, by accumulating states with forward and backward reachability computations. In this case, we can use the entire set of states $C_1 \cup C_2$ as the seed to compute C, as opposed to using a single state in $C_1 \cup C2$. Hence, we can avoid the recomputation of C_1 and C_2.

6 Composition policies

The SCC refinement algorithm described in Section 3 is generic, because it does not specify:

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

GENERIC-REFINEMENT$(\mathcal{A}, L)\{$  // Concrete and abstract models
    var Work: set of Entry;
    Work = $\{(Q, \emptyset, \text{strong})\}$

    while (Work $\neq \emptyset$) {
        Pick an entry $E = (S, L', s)$ from Work;
        Choose $\mathcal{A}' \in L$ such that there is no $\mathcal{A}'' \in L'$ with $\mathcal{A}'' \leq \mathcal{A}'$;
        Over-approximate reachability computation on $\mathcal{A}'$;
        if (ENDGAME$(S, s)$) {
            if (MODEL-CHECK$(\mathcal{A}, S, s)$)
                return true;
        }
        else {
            $\mathcal{C}$ = SCC-DECOMPOSE$(S, \mathcal{A}')$;
            if ($\mathcal{C} \neq \emptyset$ and $\mathcal{A}' = \mathcal{A}$)
                return true;
            else {
                for (all $C \in \mathcal{C}$) {
                    $s$ = ANALYZE-STRENGTH$(C, \mathcal{A}')$;
                    insert $(C, L' \cup \{\mathcal{A}'\}, s)$ in Work;
                }
            }
        }
    }
    return false;
}

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

Fig. 4. The generic SCC-refinement algorithm

1. what set of over-approximations 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;
4. the criterion used to decide when to switch to the endgame.

These four aspects make up a *policy*, and are the subjects of this section. Implementation details are deferred to Section 9.

6.1 Choice of the abstract systems

We assume that $\mathcal{A}$ is the composition of a set of submodules $M = \{M_1, \dots, M_m\}$, and that the set L of over-approximations consists of the compositions of subsets of M:

$$L \subseteq \{M_{j_1} \parallel \cdots \parallel M_{j_p} \mid \{j_1, \dots, j_p\} \subseteq \{1, \dots, m\}\} \enspace .$$

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, $m \le r = |V|$. If an over-approximation $\mathcal{A}'$ has a strict subset of submodules, and $\eta_{\mathcal{A}}$ and $\eta_{\mathcal{A}'}$ are the numbers of states in $\mathcal{A}$ and $\mathcal{A}'$ respectively, we have $2\eta_{\mathcal{A}'} \le \eta_{\mathcal{A}}$.

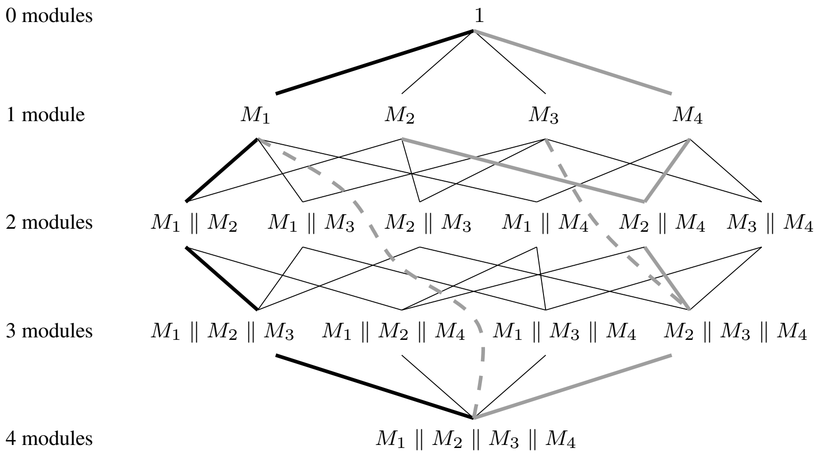

0 modules
1 module
2 modules
3 modules
4 modules
1
M_1
M_2
M_3
M_4
$M_1 \parallel M_2$
$M_1 \parallel M_3$
$M_2 \parallel M_3$
$M_1 \parallel M_4$
$M_2 \parallel M_4$
$M_3 \parallel M_4$
$M_1 \parallel M_2 \parallel M_3$
$M_1 \parallel M_2 \parallel M_4$
$M_1 \parallel M_3 \parallel M_4$
$M_2 \parallel M_3 \parallel M_4$
$M_1 \parallel M_2 \parallel M_3 \parallel M_4$

Fig. 5. Lattice of over-approximations

The set of all over-approximations generated from subsets of M forms a lattice under the relation $\le$, shown in Fig. 5 for $m = 4$. 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 m, it is impractical to make use of all 2^m over-approximations; consequently, we shall only consider efficient policies, in which any given state is contained in the SCC-closed set passed to SCC-DECOMPOSE $O(r)$ times.

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. $|L_i| \leq \lambda$, that is, each subset of L contains at most λ over-approximations;
2. for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$;
3. $\mathcal{A} \in L_r$.

Two such cases are illustrated in Fig. 5. In both cases, $(j_1, \ldots, j_m)$ is a permutation of $(1, \ldots, m)$ that identifies a linear order of the modules. The first one is called the *popcorn-line* policy. The approximations are:

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

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 left of Fig. 5 (solid thick lines), the algorithm uses a *popcorn-line* policy with $(j_1, \ldots, j_4) = (1, 2, 3, 4)$ and $\lambda = 1$. The approximations are:

$$\begin{aligned}
\mathcal{A}_1 &= M_1, \\
\mathcal{A}_2 &= M_1 \parallel M_2, \\
\mathcal{A}_3 &= M_1 \parallel M_2 \parallel M_3, \\
\mathcal{A}_4 &= M_1 \parallel M_2 \parallel M_3 \parallel M_4.
\end{aligned}$$

Another one is called the *lightning-bolt* policy. The approximations are

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

When an entry $E = (S, L', s)$ is retrieved from Work, among the two $\mathcal{A}_i$, the one with lower index is chosen first.

At the right of Fig. 5 (thick grey lines), $2n - 1$ approximations are used in a *lightning-bolt* policy, for which $(j_1, \ldots, j_4) = (4, 3, 2, 1)$ and $\lambda = 2$:

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

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.

6.2 Adaptive popcorn policy

It may be impractical to adopt the popcorn-line policy all the way down to the exact system:

1. There may be too much overhead in analyzing all the abstract models;
2. if an SCC-closed set becomes weak or terminal, checking it directly in the concrete model may be cheap.

In these two cases, one may decide to switch to the endgame. That is, after spending a reasonable amount of effort on decomposing the SCCs in the abstract models, we jump to the concrete model. When the endgame comes, there are different ways of jumping to the exact system—all of them can be considered as variants of the popcorn-line policy.

The first variant is to go to $\mathcal{A}$ directly, and search for fair cycles inside each SCC-closed set S in Work. Both SCC hull and SCC enumeration algorithms can be used for the fair-cycle detection. Assume, for instance, that $\mathcal{A}$ is the composition of the set of submodules $\{M_1, ..., M_8\}$, and we decide to jump after composing the first three submodules. This variant can be described as follows:

$$\begin{aligned}
\mathcal{A}_1 &= M_1, \\
\mathcal{A}_2 &= M_1 \parallel M_2, \\
\mathcal{A}_3 &= M_1 \parallel M_2 \parallel M_3, \\
\mathcal{A}_4 &= M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8.
\end{aligned}$$

Alternatively, one can further trim the SCC-closed sets before searching for fair cycles in the concrete model. Remaining submodules are applied, one at a time, to further partition these SCC-closed sets. This variant, called the “Cartesian product” approach, is characterized as follows:

$$\begin{aligned}
\mathcal{A}_1 &= M_1, \\
\mathcal{A}_2 &= M_1 \parallel M_2, \\
\mathcal{A}_3 &= M_1 \parallel M_2 \parallel M_3, \\
\mathcal{A}_4 &= M_4, \\
&\dots \\
\mathcal{A}_8 &= M_8, \\
\mathcal{A}_9 &= M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8.
\end{aligned}$$

Given the fact that each submodule M_i is relatively small and we consider them one at a time, the calls to SCC-DECOMPOSE are cheap. In fact, the partition of the state space into the last SCC-closed sets before jump has been based on the assumption that the state variables of other submodules are *free* variables (i.e., submodules do not affect others through these variables); by calling SCC-DECOMPOSE on these remaining modules individually, their state variables are constrained, resulting in further partition of the SCC-closed sets. A direct analogy can be observed between this approach and the MBM (Machine by Machine) algorithm of [8] in computing the set of approximate reachable states.

The third variant, called the "one-step further composition" approach, is characterized as follows:

$$\begin{aligned}
\mathcal{A}_1 &= M_1, \\
\mathcal{A}_2 &= M_1 \parallel M_2, \\
\mathcal{A}_3 &= M_1 \parallel M_2 \parallel M_3, \\
\mathcal{A}_4 &= \mathcal{A}_3 \parallel M_4, \\
&\dots \\
\mathcal{A}_8 &= \mathcal{A}_3 \parallel M_8, \\
\mathcal{A}_9 &= M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8.
\end{aligned}$$

The second variant does not compose prior to making the full jump; in contrast, the "one-step-further" approach invests more heavily 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. For a transition to exist in the composition, it must exist in both of the machines being composed. Whereas the second variant never fractures SCCs by this joint constraint (since it assumes submodules do not affect each other), the third variant does, ultimately leading to the partitioning of these SCCs into smaller SCC-closed sets.

Following this line to the extreme would lead us all the way back to the original popcorn-line policy, in which the fully iterative composition is used. The first variant represents the least investment, and therefore suffers the least amount of overhead. However, when it performs the most expensive part of the computation—cycle detection in the exact system—it must search a large state subspace. Conversely, the fully iterative approach has the smallest state subspace to search, but incurs the greatest overhead.

6.3 SCC refinement trees

The popcorn-line approach defines an SCC refinement tree like the one of Fig. 6 that highlights the potential advantage 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, which are the property automaton and the eight modules for the philosophers. The property passes, i.e., no fair cycles exist in the system. Only the nodes representing fair SCCs are shown in this tree. The nodes at Level i are the fair SCCs of $\mathcal{A}_i$, together with their numbers of states. ($\mathcal{A}_1$ is the property automaton.) Reachability analysis shows that there are about 47k reachable states. Note that only very small sets of states remain after the composition of the first four modules—the property automaton, the philosopher named in the property, and her two neighbors—and that no work is done on the exact system.

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. 6, 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 may. 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, since at any time, only one over-approximation (and its state transition graph) need to be loaded into the main memory.

7 Disjunctive decomposition

After switching from the adaptive popcorn policy to the endgame, we need to search for fair cycles on the exact system. This can be done by computing $\mathsf{EF}\, S$, $\mathsf{EF}\,\mathsf{EG}\, S$, or $\mathsf{EG}_{\mathsf{fair}}\, S$, depending on the strength of each individual set. We can conduct the search

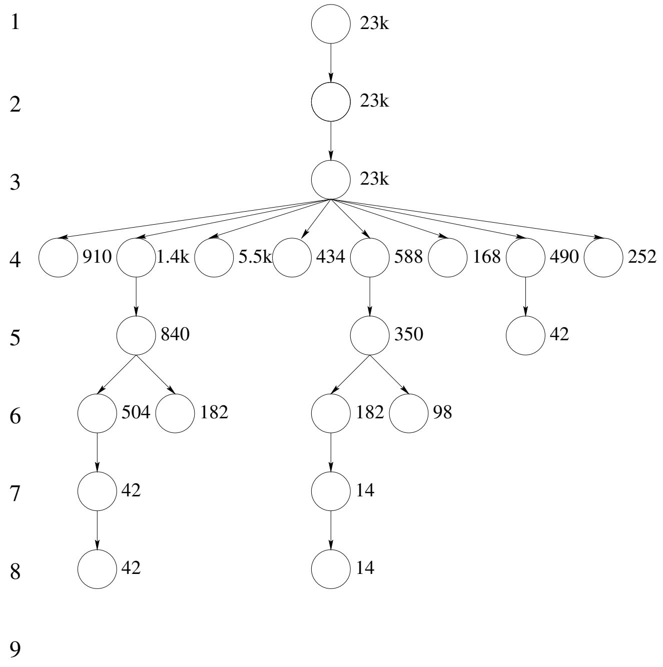


Fig. 6. SCC refinement tree

either by considering all the fair SCC-closed sets simultaneously, or by considering each of them in sequence. When the exact system is complex, however, considering each of them separately may be the only feasible approach.

7.1 Decomposition of the exact system

Theorem 3 allows us to disjunctively decompose the exact system into subautomata $\{\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}\}$, where $\mathcal{G}_i^{C_j}$ is an initial-fair path in the SCC quotient graph of $\mathcal{A}'$. Since each of these subgraphs corresponds to a depth-first search path in the SCC graph, containing a set of abstract counter-examples, it is called a *hyperline*. Our algorithm goes through all these hyperlines and checks language emptiness on each of them in isolation.

Computing all the hyperlines requires not only all fair SCCs of $\mathcal{A}'$, but also the non-fair SCCs. These non-fair SCCs can be computed with SCC-DECOMPOSE, and just like the fair ones, they can also be computed incrementally. As one may expect, the number of hyperlines in an SCC graph—a DAG—is exponential in the size of the graph. In order to avoid an excessive partitioning cost on the over-approximations, with the consequent exponential number of hyperlines, we apply the following heuristic control on the size of the SCC graphs:

- Skip applying SCC-DECOMPOSE on S if S is a non-fair SCC-closed set and its size is below a certain threshold.
- Switch to the endgame if the number of edges of the SCC graph exceeds a certain threshold.
- Switch to the endgame if the number of fair nodes of the SCC graph exceeds a certain threshold.

With such a heuristic control, the number of hyperlines is bounded by a constant value.

7.2 Guided search for fair cycles

In the endgame, we disjunctively decompose the exact state space into subspaces according to the different hyperlines of the last abstract model. Every hyperline, or every $\mathcal{G}_i^{C_j}$ subgraph, induces a subautomaton of the exact system. Language emptiness is checked in these subautomata in isolation. Although subautomata may share states, we can avoid visiting any state more than once by keeping a set of visited states globally and not looking at them again. Within $\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}$, we search for cycles that are both reachable and fair. Although reachability analysis shares the same worst-case complexity bound with the best cycle-detection algorithm, in practice it is usually much cheaper than fair cycle detection. This motivates us to always make sure a certain state subspace is reachable before deploying the cycle detection procedure, in order to avoid searching unreachable states for a fair cycle.

Fair cycle detection is triggered only after the reachability analysis hits one or more *promising* states—states that are in fair SCC-closed sets and at the same time satisfy some acceptance conditions. Recall that the symbolic SCC enumeration algorithms [2, 3, 15] used in SCC-DECOMPOSE compute an SCC by first picking a *seed*. The promising states encountered during forward reachability analysis are good candidates for the seed. In addition to the order in which they are encountered during the forward search, promising states are further prioritized according to the number of acceptance conditions they satisfy: if two promising states are hit simultaneously by the forward search, whichever satisfies more acceptance conditions is preferred. By prioritizing the seeds, we heuristically choose the SCC that is expected to be closer to the initial states and more likely to be fair; this may reduce the number of reachable states traversed by forward search and may lead to a shorter counter example. Note that the algorithm in [17] was also designed to avoid visiting too many reachable states in the search for fair cycles, although their approach was significantly different from ours.

Despite our disjunctive decomposition, there may still be many reachable states within each subautomaton. The ideal way of finding a fair cycle, if it exists, is to traverse only part of the reachable states of the subautomaton, and go directly to a promising state to start the SCC enumeration. To reach a promising state with the least possible overhead (i.e., traversing the least number of reachable states), we need some guidance for such a targeted search.

The intermediate results of the reachability analysis of $\mathcal{A}'$, can provide a guidance for the targeted search. Reachability analysis with Breadth-First Search (BFS) gives us the set of *reachable onion rings*, denoted by $\{R^0, R^1, \ldots, R^l\}$; each ring is the set of states at a certain distance from the initial states. For example, a state in R^2 cannot be reached from an initial state in less than two steps. Suppose that the promising state nearest to the initial states is in R^3, one wants to spend as little effort as possible in traversing states in R^1 and R^2.

The targeted search algorithm DETECT-FAIR-CYCLE, given in Fig. 7, is called by MODEL-CHECK for every hyperline $\mathcal{G}'$. The two global variables $Reach$ and $Queue$ represent the set of reached states and the SCC-closed sets, respectively. The abstract reachable onion rings of $\mathcal{A}'_{sub}$, denoted by $absRings$ or $\{R^i\}$, are used to estimate the distance of SCCs to the initial states. The distance is used to order the SCC-closed sets in the priority queue $Queue$. Following this order, we pick up the SCC sets one by one inside SCC-DECOMPOSE-WITH-EARLYTERMINATION and search for fair cycles. The procedure DETECT-FAIR-CYCLE terminates when either all reachable states in $\mathcal{A}_{sub}$ are visited, or a fair cycle is found.

Instead of using the conventional IMAGE computation, we use a heuristic algorithm called *sharp image* computation for the targeted reachability analysis. The pseudocode for the sharp image computation is also given in Fig. 7. Let D be the*from set*, and $\{R^i\}$ be the set of reachable onion rings from an abstract model. The procedure first finds the ring that is closest to the target and at the same time intersects D. The intersection of this ring and D is further compacted by BDD-SUBSETTING [33, 31], resulting in $D^\#$, whose states have the shortest approximate distance to the promising states. As a generic function, BDD-SUBSETTING returns a minterm, a cube, or an arbitrary subset of $(D \cap R^i)$ with a small BDD representation. Finally, the image of $D^\#$ is computed by the conventional IMAGE operation; such an image is obviously a subset of $\mathsf{EY}(D)$.

Although related to it, our sharp search is different from the high-density algorithm of [33], because our goal in compacting the from set is to get closer to the fair SCCs, not increase the density of its BDD representation. Nevertheless, our approach has a problem in common with high-density search—namely, how to recover from *dead-ends*. Since IMAGE$^\#$ computes only a subset of the exact image, it is possible for the frontier set, $Front$, to be empty before the forward search actually reaches the fix-point. Whenever this happens, we need to backtrack with the standard IMAGE.

Every time a promising state is encountered during the targeted reachability analysis, it is picked as a seed for computing the SCC. The entire search terminates if the SCC containing the seed is fair. If the SCC is not fair, it is merged into the set of already reached states $Reach$ before the targeted reachability analysis is resumed (because the SCC has proved to be reachable). Since every SCC found in this way is guaranteed to be reachable, the SCC enumeration algorithms [2, 3, 15] can be further enhanced with early termination [34]: they terminate as soon as a fair cycle is found, as opposed to after both the forward and backward search from the seed reach their fix-points. This requires that after each forward and backward step, we check whether the intersection of the forward and backward results satisfies all the fairness conditions—if it does, the union of the forward and backward search results contains a reachable fair cycle.

```
DETECT-FAIR-CYCLE($\mathcal{A}, \mathcal{A}', \mathcal{G}', Reach, Queue$){    // model, abs model, hyperline,
                                                                              // reached states, and scc-closed sets
    $\mathcal{A}'_{sub} = \mathcal{A}' \Downarrow \mathcal{G}'$;
    $\mathcal{A}_{sub} = \mathcal{A} \Downarrow \mathcal{G}'$;

    $absRings$ = COMPUTE-REACHABLE-ONIONRINGS($\mathcal{A}'_{sub}$);
    $Front = Reach$;

    while (true) {

        while ($Front \neq \emptyset$) and ($Front \cap Queue = \emptyset$) {
            $Front$ = IMAGE$^{\#}$($\mathcal{A}_{sub}, Front, absRings$) $\setminus Reach$;
            if ($Front = \emptyset$)
                $Front$ = IMAGE($\mathcal{A}_{sub}, Reach$) $\setminus Reach$;
            $Reach = Reach \cup Front$;
        }

        if ($Front = \emptyset$)
            return false;

        if (SCC-DECOMPOSE-WITH-EARLYTERMINATION($\mathcal{A}_{sub}, Queue, absRings$))
            return true;
    }
}

IMAGE$^{\#}$($\mathcal{A}, D, \{R^i\}$) {

    $i = |\{R^i\}|$;
    while ($D \cap R^i = \emptyset$) {
        $i = i - 1$;
    }

    $D^{\#}$ = BDD-SUBSETTING ($D \cap R^i$);

    return IMAGE($\mathcal{A}, D^{\#}$);
}
```

Fig. 7. The guided search for fair cycles and *sharp* image computation algorithm

When the language is indeed empty, all reachable states of the subautomata must be traversed. Let $\eta_\mathcal{A}$ be the number of reachable states of the exact system, and let the total number of hyperlines be a constant value; then, the cost of deciding reachability is $O(\eta_\mathcal{A})$. The total cost of the targeted search depends on the underlying symbolic SCC enumeration algorithm, which we analyze in the next section.

8 Complexity

The refinement algorithm described thus far cannot improve the worst-case complexity of the language emptiness check. In the following, we show that the complexity of our incremental approach is within a constant factor from that of the non-incremental one; this means that it is $O(\eta_\mathcal{A})$ if [15] is used in SCC-DECOMPOSE, or $O(\eta_\mathcal{A} \log \eta_\mathcal{A})$ if [2, 3] is used instead. In the following theorem, we assume that the algorithm of [15] is used.

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 $O(\eta_\mathcal{A})$ steps.

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

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

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

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 linear complexity result—even when the $n \log n$ algorithm in [2, 3] is used.

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 $O(\eta_{\mathcal{A}})$. □

9 Implementation and experiments

9.1 Compositional SCC refinement

First, we describe the 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 approach, with breadth-first search of the SCC refinement tree. The set of over-approximated models are generated according to the strategy of [18], which divides the set of state variables (or latches) into many small clusters. These clusters are considered as the submodules, whose composition is the exact system. The submodules are sorted according to their distances from the state variables appearing in 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. 5 and 6. To avoid too much overhead in the analysis of the over-approximations, it heuristically stops the refinement at some point, and then immediately composes all the remaining modules, thus proceeding directly to the exact system. In the implementation, we stop the linear composition after 30% of the state variables have been composed to avoid having too many fair SCCs in the full SCC refinement tree. Once the exact system is reached, the Emerson-Lei algorithm is applied to its SCC-closed sets. 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 fair SCCs exist, the algorithm checks their strength. All the weak SCCs 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 SCC exists, the algorithm also terminates: there is no fair cycle. Otherwise, the approximate system is refined.

Our algorithm was implemented in the symbolic model checker VIS [5, 37], and the results of Table 1 were obtained by appropriately calling the standard Language Emptiness command of VIS. SCC analysis was performed with the Lockstep algorithm of [2, 3]. (A separate study showed that the algorithm of [15], in spite of its better worst-case bound, in practice has a performance slightly worse than Lockstep.) Prior reachability analysis results were used as don't cares where possible. 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 set of results, with dynamic variable ordering turned on for each example. Similarly, we obtained a third set of results, using the EL2 variant of the Emerson-Lei algorithm [17, 14, 34]. These second and third sets are omitted, since their character is not significantly different. (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.) The experiments were run on an IBM Intellistation running Linux with a 400MHz Pentium II processor with 1GB of RAM.

Table 1 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 state variables in the system. The three fields of the second column, obtained by direct application of the Emerson-Lei algorithm as implemented in VIS, give:

1. the time it took to run the experiment (Time/Out (T/O) indicates a run time greater than 4 hours);
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.

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 the algorithm found a weak fair SCC containing a fair cycle.

The property automata being used in the experiment were 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 an n or w in the last field demonstrates that both pruning of the SCC refinement tree and strength reduction are effective in these experiments.

Table 1. Comparing Emerson-Lei and D‘n’C. With RDCs.

			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 (s)	BDD (M)	EX/EY	
bakery1	F	56	212	5.1	5337/0	31	1.3	354/4	27	1.3	484/328	
bakery2	P	49	69	3.4	526/0	20	1.3	10/4	20	1.3	62/73	n
bakery3	P	50	421	14	1593/0	46	2.5	90/4	43	1.8	537/428	
bakery4	F	58	T/O	-	-/-	1950	3.4	1088/5	1337	4.7	947/96	
bakery5	F	59	T/O	-	-/-	1009	6.1	127/5	623	6.1	216/243	
eisenb1	F	35	23	1.0	416/0	16	0.9	21/4	16	0.9	21/4	
eisenb2	F	35	T/O	-	-/-	4800	8.2	162/5	1683	7.7	105/93	w
elevator1	F	37	210	14	163/0	49	2.8	132/9	41	2.2	155/31	
nmodem1	P	56	4384	11	5427/0	192	1.1	992/4	233	0.6	5007/71	
peterson1	F	70	17	1.1	24/0	20	1.3	19/4	21	1.2	157/173	
philo1	F	133	371	12	258/0	7	0.2	8/12	7	0.2	8/12	w
philo2	F	133	73	2.8	557/0	30	1.3	258/5	12	0.5	25/44	w
philo3	P	133	T/O	-	-/-	T/O	-	-/-	115	1.2	993/224	
shamp1	F	143	44	2.1	8/0	103	5.6	9/6	87	2.2	266/280	
shamp2	F	144	T/O	-	-/-	1892	16.	74/6	101	2.9	345/349	
shamp3	F	145	T/O	-	-/-	337	4.4	19/17	335	4.4	19/17	w
twoq1	P	69	12	0.4	25/0	4	0.1	7/9	4	0.1	7/9	n
twoq2	P	69	241	8.9	175/0	27	0.8	91/5	30	0.9	181/95	

Comparing the D‘n’C algorithm to the one by Emerson and Lei, we find, with only three exceptions out of 18 examples, 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 limited locality. We expect the compositional algorithm to do even better on models with more locality. On the other hand, one can see that the greater compositionality of the Multi-Level version proves its worth, especially on the larger examples.

9.2 Disjunctive decomposition and guided search

Here, we describe the details of another implemented policy for the disjunctive decomposition and the sharp guided search algorithm in our general framework. We call the enhanced algorithm D‘n’C$^{\#}$, which uses the adaptive popcorn-line approach to trim the fair SCC-closed sets before we jump to the exact system (by the “Cartesian Product” approach). We then use guided fair cycle detection to inspect each hyperline restricted subautomaton. We compared D‘n’C$^{\#}$ with the original D’n’C algorithm on the same set of test cases to demonstrate the effectiveness of the added features. The results are shown in Tables 2 and 3, whose data were obtained on a 400MHz Pentium II processor with 1GB of RAM.

In Table 2, prior reachability analysis results were used as don’t cares where possible. 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 or fails, and the number of binary state variables in the system. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D‘n’C$^{\#}$ algorithm to D‘n’C, we find three wins for D‘n’C$^{\#}$ and 15 wins for D‘n’C. This indicates that the disjunctive decomposition is encumbered by overhead of maintaining and decomposing the SCC graph. However, among the 15 wins of D‘n’C, only four are for problems requiring more than 100 seconds to complete—that is, the easy problems. In contrast, on the three wins of D‘n’C$^{\#}$, D’n’C took 1337, 1683, and 233 seconds. Therefore, we conclude that the additional overhead is not significant, and on the harder problems, D‘n’C$^{\#}$ is as competitive as D‘n’C when advance reachability analysis is feasible.

In Table 3, the same set of test cases were checked with the approximate reachability analysis results as the don’t cares where possible—that is, with ARDCs as opposed to RDCs. Note that if RDC is already available, there is no need to do reachability on each level in D‘n’C. However, RDC itself is in general very expensive to compute; when RDC is not available, language emptiness becomes a much hard problem. (Approximate reachability analysis is usually much faster than exact reachability analysis, and in practice, may be the only feasible way of extracting don’t cares from reachable states.) The table has four columns. The three fields of the first column repeat the description of

Table 2. Comparing EL, D‘n’C, and D’n’C$^{\#}$. With RDCs.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/F	latch num	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$
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

Table 3. Comparing EL, D‘n’C, and D‘n’C$^{\#}$. With ARDCs.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/F	latch num	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$
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
elevator1	F	37	3504	2156	585	663	612	657	24.4	21.1	23.6
nmodem1	P	56	T/O	T/O	3375	-	-	306	-	-	2.6
peterson1	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
shamp1	F	143	12	T/O	168	21	-	127	0.0	-	2.0
shamp2	F	144	T/O	T/O	189	-	-	153	-	-	3.0
shamp3	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

Table 4. Comparing EL, D‘n’C, and D‘n’C$^{\#}$. With ARDCs.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/F	latch num	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$	EL	D‘n’C	D‘n’C$^{\#}$
Blackjack1	F	176	7296	2566	237	618	610	551	26.8	24.2	18.1
MSI1	P	65	T/O	T/O	51	-	-	83	-	-	2.0
MSI2	F	65	T/O	T/O	165	-	-	342	-	-	6.7
PIbus1	P	387	T/O	73	1700	-	243	539	-	3.5	13.4
PIbus2	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

the test cases. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D‘n’C$^{\#}$ algorithm to D‘n’C, we find 12 wins for D‘n’C$^{\#}$ and six for D‘n’C. In addition, all the four wins for D‘n’C are for problems requiring less than 100 seconds to complete; in contrast, D‘n’C$^{\#}$ wins more on the harder ones—on eight out of its 12 wins, D‘n’C timed out after 4 hours. The difference here is that both D‘n’C and EL depend heavily on full reachability to restrict the search spaces, but the disjunctive decomposition and sharp guided search of D‘n’C$^{\#}$ minimizes this dependency.

We also conducted the experiments on a set of much harder test cases, the Texas-97 benchmark circuits. The property automata being used in the experiment were also translated from LTL formulae. The experiments were run on an IBM Intellistation with a 1700MHz Pentium-IV processor and 2GB of SDRAM. The result is given in Table 4.

In Table 4, the comparison is with the results of approximate reachability analyses as the don’t cares where possible. Note that exact reachability analysis is infeasible for most of these circuits, except for MSI. 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 or fails, and the number of binary state variables in the system. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D‘n’C$^{\#}$ algorithm to D‘n’C, we find five wins for D‘n’C$^{\#}$ and two for D‘n’C. Again, the two wins for D‘n’C are easier problems, and the five wins for D‘n’C$^{\#}$ are much harder—among them, two cannot be finished by D‘n’C within 8 hours. Therefore, it demonstrates a decisive advantage of the D‘n’C$^{\#}$ algorithm over both D‘n’C and EL.

10 Conclusions

In this paper, we have shown that over-approximations of the exact system can be used to gradually refine the SCC-closed sets to its SCCs, and have presented a general framework for SCC refinement. 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 discussed different policies in traversing the lattice of over-approximated systems, and have implemented two of them. In comparison to the original Emerson-Lei algorithm, our experimental results demonstrate significant and almost consistent performance improvement. This indicates the importance of the three improvement factors built into our algorithm: (1) SCC refinement, (2) compositionality, and (3) strength reduction.

We have also shown that the SCC quotient graph of the over-approximated system can be used to decompose the concrete search state space. Based on the disjunctive decomposition, we have presented a targeted search algorithm for fair cycle detection, with the approximate distance to fair SCCs as the guidance. Experiments have shown that for large or otherwise difficult problems, heavy investment in these heuristics is well justified.

Our generic framework can be highly parallelized by assigning different entries from the Work list, as well as different subautomata, to different processors. Processors that deal with disjoint sets of states have minimal communication and synchronization requirements. Although the algorithm is geared towards BDD-based symbolic model checking, SCC refinement can also be combined with explicit state enumeration and SAT-based approaches.

The simplicity of the implemented policies in comparison to the generality of our framework suggests that there can be many promising extensions and variations. In particular, the way in which the set of over-approximations are generated is still primitive—it is based solely on the structual information of the circuit. In this sense, the various counter-example guided refinement techniques [21, 18, 9, 41, 10, 7, 40] proposed in abstraction refinement can be incorporated into our algorithm generic framework. The joint application of over- and under-approximations of the exact system can also be interesting future work.

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV '93)*. Springer-Verlag, Berlin, 1993. LNCS 697.

[2] 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.

[3] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. *Formal Methods in System Design*, 27(2), September 2005. To appear.

[4] 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.

[5] 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.

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

[7] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, November 2002. LNCS 2517.

[8] 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.

[9] 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. LNCS 1855.

[10] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Ap-*

plied Formal Methods for Correct VLSI Design, pages 111–128, Leuven, Belgium, November 1989.

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

[13] 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.

[14] K. Fisler, R. Fraer, G. Kamhi, M. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434. Springer-Verlag, April 2001. LNCS 2031.

[15] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected componenets in a linear number of symbolic steps. In *Symposium on Discrete Algorithms*, Baltimore, MD, January 2003.

[16] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 416–423, November 2003.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, San Jose, CA, November 1996.

[23] B. Li, C. Wang, and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *Software Tools for Technology Transfer*, 2(7):143–155, 2005.

[24] 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.

[25] J. Lind-Nielsen, H. R. Andersen, G. Behrmann, H. Hulgaard, K. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, pages 201–216, Lisbon, Portugal, March-April 1998. LNCS 1384.

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

[27] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, April 2003. LNCS 2619.

[28] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[29] 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.

[30] A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 12–23. Springer-Verlag, Berlin, 1997. LNCS 1254.

[31] A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference*, pages 457–462, San Francisco, CA, June 1998.

[32] 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.

[33] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995.

[34] F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 88–105. Springer-Verlag, November 2002. LNCS 2517.

[35] 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.

[36] 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.

[37] URL: http://vlsi.colorado.edu/$\sim$vis.

[38] 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.

[39] C. Wang and G. D. Hachtel. Sharp disjunctive decomposition for language emptiness checking. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 105–122. Springer-Verlag, November 2002. LNCS 2517.

[40] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–415, November 2003.

[41] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.

[42] 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.