Advanced Automata-based Algorithms for Program Termination Checking

Yu-Fang Chen  
Academia Sinica, Taiwan  
National Taipei University, Taiwan  
yfc@iis.sinica.edu.tw

Matthias Heizmann  
University of Freiburg, Germany  
heizmann@informatik.uni-freiburg.de

Ondřej Lengál  
FIT, Brno University of Technology  
IT4Innovations Centre of Excellence  
Czech Republic, lengal@fit.vutbr.cz

Yong Li  
State Key Laboratory of Computer Science, Institute of Software  
Chinese Academy of Sciences  
University of Chinese Academy of Sciences, China, liyong@ios.ac.cn

Ming-Hsien Tsai  
Academia Sinica, Taiwan  
mhtsai208@gmail.com

Andrea Turrini  
State Key Laboratory of Computer Science, Institute of Software  
Chinese Academy of Sciences  
China  
turrini@ios.ac.cn

Lijun Zhang  
State Key Laboratory of Computer Science, Institute of Software  
Chinese Academy of Sciences  
University of Chinese Academy of Sciences, China, zhanglj@ios.ac.cn

Abstract

In 2014, Heizmann et al. proposed a novel framework for program termination analysis. The analysis starts with a termination proof of a sample path. The path is generalized to a Büchi automaton (BA) whose language (by construction) represents a set of terminating paths. All these paths can be safely removed from the program. The removal of paths is done using automata difference, implemented via BA complementation and intersection. The analysis constructs in this way a set of BAs that jointly “cover” the behavior of the program, thus proving its termination. An implementation of the approach in Ultimate Automizer won the 1st place in the Termination category of SV-Comp 2017.

In this paper, we exploit advanced automata-based algorithms and propose several non-trivial improvements of the framework. To alleviate the complementation computation for BAs—one of the most expensive operations in the framework—, we propose a multi-stage generalization construction. We start with generalizations producing subclasses of BAs (such as deterministic BAs) for which efficient complementation algorithms are known, and proceed to more general classes only if necessary. Particularly, we focus on the quite expressive subclass of semideterministic BAs and provide an improved complementation algorithm for this class. Our experimental evaluation shows that the proposed approach significantly improves the power of termination checking within the Ultimate Automizer framework.

CCS Concepts  • Theory of computation → Automata over infinite objects; • Software and its engineering → Formal software verification;

Keywords  Program Termination, Büchi Automata Complementation and Language Difference

ACM Reference Format:

1 Introduction

Termination analysis of programs is a challenging area of formal verification, which has attracted the interest of many researchers approaching the problem from different angles [4, 13, 17–19, 27, 28, 31, 33, 34, 36, 38, 42, 45–48, 51, 52]. All approaches need to deal with the following challenge: when a program contains loops with branching or nesting, how to
devise a termination argument that holds for any possible interleaving of different paths through the loop body?

Due to the difficulty of solving the general problem, many researchers have focused on its simplified version that addresses only lasso-shaped programs, i.e., programs where the control flow consists of a stem followed by a simple loop without any branching. Proving termination of this class of programs can be done rather efficiently [6, 8, 9, 14, 16, 32, 39, 44].

The approach of Heizmann et al. [33] leverages those results and proposes a modular construction of termination proofs for a general program \( P \) from termination proofs of lasso-shaped programs obtained from its concrete paths. On a high level, the approach repeatedly performs the following sequence of operations (see Figure 1). First, it samples a path \( \tau = u \omega \) from the possible behaviors of \( P \) and attempts to prove its termination using an off-the-shelf approach. Second, it generalizes \( \tau \) into a (potentially infinite) set of paths \( M \), called a certified module, that all share the same termination proof with \( \tau \). Finally, it checks whether the behavior of \( P \) contains a path \( \tau' \) not covered by any certified module generated so far and, if so, the procedure is restarted. This sequence is repeated until either a non-terminating path is found or all behaviors of \( P \) are covered by the modules.

Let us follow with an informal description of the procedure on the example program \( p_{\text{sort}} \) in Figure 2a. Figure 2b shows the control flow graph (CFG) of \( p_{\text{sort}} \) as a Büchi automaton (BA) \( \mathcal{A}_{p_{\text{sort}}} \). The alphabet of \( \mathcal{A}_{p_{\text{sort}}} \) is the set of all statements occurring in \( p_{\text{sort}} \) and each state of \( \mathcal{A}_{p_{\text{sort}}} \) is a location of \( p_{\text{sort}} \). All states of \( \mathcal{A}_{p_{\text{sort}}} \) are accepting so every infinite sequence of statements of the program corresponds to an infinite word in \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}}) \). The task is to decompose \( \mathcal{A}_{p_{\text{sort}}} \) into a finite set of Büchi automata \( \mathcal{A}_1, \ldots, \mathcal{A}_n \), each representing a program with a termination argument, such that \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \cdots \cup \mathcal{L}(\mathcal{A}_n) \), so every path of \( p_{\text{sort}} \) is represented by a word in \( \mathcal{A}_{p_{\text{sort}}} \) and is guaranteed to terminate by an argument for some \( \mathcal{A}_i \). More concretely, each BA \( \mathcal{A}_i \) is associated with a ranking function \( f_i \) and a rank certificate \( I_i \), mapping each state to a predicate over the program variables (cf. Section 3). The triple \( M_i = (\mathcal{A}_i, f_i, I_i) \) is called a certified module. The construction of the set \( \{M_1, \ldots, M_n\} \) is step-wise (see Figure 1). First, we find an ultimately periodic word \( u \omega \in \mathcal{L}(\mathcal{A}_{p_{\text{sort}}}) \) which is essentially a lasso-shaped program and use a standard approach to check if it corresponds to a terminating path. In our example, we start with sampling the word \( u \omega = 1>0, j:=1, w \). We can prove termination of the path corresponding to \( u \omega \) by finding, e.g., the ranking function \( f_1(i, j) = 1 - j \).

In the following, we will denote the outer loop of \( \mathcal{A}_{p_{\text{sort}}} \) as \( \text{Outer} = j:=1, i-- \) and its inner loop as \( \text{Inner} = j<i, j++ \). We can observe that \( f_1 \) is also a ranking function for the set of paths obtained by generalizing \( u \omega \) into the set of words that correspond to all paths that eventually stay in the inner loop, i.e., words from \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1) = 1>0, j:=1 \). The language \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1) \) together with a ranking function \( f_1 \) and a rank certificate \( I_1 \) can be represented by the certified module \( M_1 = (\mathcal{A}_1, f_1, I_1) \) where \( \mathcal{L}(\mathcal{A}_1) = L_1 \). We proceed by removing all paths covered by \( L_1 \) from \( \mathcal{A}_{p_{\text{sort}}} \) to know which paths still need to be examined. The removal can be performed by executing a BA difference algorithm, followed by checking language emptiness (potentially finding a new counterexample \( u \omega \) on failure). In our example, the difference corresponds to the (non-empty) language \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1, \mathcal{A}_2) = 1>0, j:=1 \cdot (\text{Inner} \cdot \text{Outer})^\omega \).

The language \( L_2 \) obtained after removing the words covered by \( L_1 \) can be represented by \( \mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1, \mathcal{A}_2 \). We denote the inner loop by \( \text{Inner} \cdot \text{Outer}^\omega \). Suppose the next sampling gives us \( u \omega = 1>0, j:=1 \cdot \text{Outer}^\omega \), for which, e.g., the ranking function \( f_2(i, j) = 1 \) is applicable. Note that \( f_2 \) is also a valid ranking function for all paths taking the outer while loop infinitely often, i.e., all paths corresponding to words from \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1, \mathcal{A}_2) \). We represent these paths by the certified module \( M_2 = (\mathcal{A}_2, f_2, I_2) \) where \( \mathcal{L}(\mathcal{A}_2) = L_2 \). After removing the words from \( \mathcal{L}(\mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1, \mathcal{A}_2) \), we, finally, obtain the BA \( \mathcal{A}_{p_{\text{sort}}} \mid \mathcal{A}_1, \mathcal{A}_2 \) whose language is empty. This means that the modules \( M_1 \) and \( M_2 \) cover all possible paths of the program \( p_{\text{sort}} \) and, because each of them comes with a termination argument, we can conclude that all paths of \( p_{\text{sort}} \) are guaranteed to terminate.

Figure 1. The flow of the automata-based termination analysis and the scope of the paper.

Figure 2. An example program and its BA representation.
Note that the above procedure performs extensively computation of language difference of a pair of BAs. The computation of difference involves computing the complement of a BA, one of the most difficult operations in automata theory—it is known that complementing a BA with \( n \) states has the lower-bound space complexity \( 2^{O(n \log n)} \) [40].

In this paper, we exploit advanced automata-based algorithms and propose several non-trivial improvements of the framework. Our main contributions are the following:

**Contribution 1:** We devise a multi-stage generalization approach, which tries to avoid the costly complementation of general BAs by considering several subclasses of BAs with cheaper complementation operations. For every terminating trace represented as a word \( w^{\omega} \), we consider the following subclasses: (i) finite-trace BAs (BAs accepting the language \( w^{2^{n_k}} \) for a word \( w \in \Sigma^{*} \)), (ii) deterministic BAs (DBAs), (iii) semideterminate BAs (SDBAs) where, intuitively, all strongly connected components (SCCs) containing an accepting state are deterministic, and, finally, (iv) general BAs. These subclasses indeed have more efficient complementation procedures: the complementation of finite-trace BAs needs only \( O(1) \) space, DBAs can be complemented in \( O(n) \) space [35], and complementing SDBAs requires only \( 2^{O(n)} \) space [12]. The details of the multi-stage approach are presented in Section 3. Our observation from running the multi-stage approach is that general BAs are needed only rarely—in the vast majority of cases, SDBAs are expressive enough.

**Contribution 2:** In our multi-stage approach shown above, the computation of the difference automaton for a BA and an SDBA is one of the most expensive operations in the loop of automata-based termination analysis. Our second contribution is an efficient algorithm for computing the language difference of a BA and an SDBA. The difference algorithm performs on-the-fly intersection and complement, on the top level using the (as far as we know currently the most efficient) SCC-based BA emptiness checking algorithm of [26]. The details of the algorithm are given in Section 4.

**Contribution 3:** Our third contribution is the improvement of the efficiency of state-of-the-art algorithms manipulating SDBAs. We, in particular, provide several heuristics of the SDBA complementation procedure of Blahoudek et al. from [12]. The main ideas of the heuristics are the following: (i) lazy construction, which delays nondeterministic choices in a way similar to partial order reduction [30, 43, 54] (Section 5), and (ii) subsumption-based pruning of states inspired by antichain-based algorithms for testing universality and language inclusion over finite automata [2, 23] (Section 6).

We implemented the proposed solutions in the open source tool **Ultimate Automizer** and evaluated them on the benchmarks from SV-Comp [1]. Our experimental evaluation (Section 7) shows that the approach we propose in this work has significantly improved the power of termination checking within the **Ultimate Automizer** framework.

### 2 Preliminaries

We fix an alphabet \( \Sigma \). A (nondeterministic) generalized Büchi automaton (GBA) with \( k \) accepting sets is a tuple \( \mathcal{A} = (Q, \delta, Q_I, F) \), where \( Q \) is a finite set of states, \( \delta : Q \times \Sigma \rightarrow 2^Q \) is a transition function, \( Q_I \subseteq Q \) is a set of initial states, and \( F = \{ F_j \subseteq Q \mid j \in \{1, \ldots, k\} \} \) is a set of accepting conditions. Unless stated explicitly, we assume that all GBAs are complete, i.e., for each \( q \in Q \) and \( a \in \Sigma \), it holds that \( \delta(q, a) \neq \emptyset \). We use \( \vec{q} \rightarrow p \) to denote that \( p \in \delta(q, a) \), and we define \( \text{post}(q) = \bigcup_{a \in \Sigma} \delta(q, a) \). We lift \( \delta \) to sets of states in the usual way. We abuse notation and for \( q \in Q \) use \( \mathcal{F}(q) = \{ f \in \{1, \ldots, k\} \mid q \in F_f \} \) to denote the set of accepting conditions that \( q \) satisfies. Moreover, we sometimes use \( \mathcal{F} \) also to denote the set \( \{1, \ldots, k\} \).

A trace of \( \mathcal{A} \) on an infinite word \( w = w_0w_1 \ldots \in \Sigma^\omega \) from a state \( q_0 \) is an infinite sequence of transitions \( \pi = q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \) such that for each \( i \geq 0 \), we have \( q_i \xrightarrow{w_i} q_{i+1} \). The trace \( \pi \) is accepting iff for each \( 1 \leq j \leq k \), there are infinitely many \( i \) such that \( q_i \in F_j \), and is safe iff for all \( 1 \leq i \leq 0 \), \( q_i \notin \bigcup_{j \leq k} F_j \). A run \( \rho = q_0q_1 \ldots \) is the projection of a trace to states. The concept that a run is accepting or safe is defined analogously. The language of a state \( q \in Q \) in \( \mathcal{A} \) is the set \( \mathcal{L}(q) = \{ w \in \Sigma^\omega \mid \mathcal{A} \text{ has an accepting trace on } w \text{ from } q \} \) (denoted also as \( \mathcal{L}(q) \) if \( \mathcal{A} \) is obvious). If \( \mathcal{L}(q) = \emptyset \), we call \( q \) useless. The language of the GBA \( \mathcal{A} \) is defined as \( \mathcal{L}(\mathcal{A}) = \bigcup_{q \in Q_I} \mathcal{L}(q) \). We use \( \subseteq_{L} \) to denote the relation of language inclusion of states: \( p \subseteq_{L} q \iff L(p) \subseteq L(q) \).

A Büchi automaton (BA) is a GBA with just \( k = 1 \) accepting condition, i.e., \( \mathcal{F} = \{ F \} \). We often denote a BA as \( (Q, \delta, Q_I, F) \). A complement of \( \mathcal{A} \) is a BA \( \mathcal{A}_C \) that accepts the language \( \mathcal{L}(\mathcal{A}_C) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A}) \). \( \mathcal{A} \) is a deterministic BA (DBA) if \( \forall q \in Q, a \in \Sigma : |\delta(q, a)| \leq 1 \land |Q_I| = 1 \). Moreover, \( \mathcal{A} \) is a semideterminate BA (SDBA) if, for each \( q_f \in F \), the automaton \( \mathcal{A}(q_f) \) is deterministic, where \( \mathcal{A}(q_f) \) is obtained from the BA \( (Q, \delta, q_f, F) \) by removing states unreachable from \( q_f \). Intuitively, this means that an SDBA can move nondeterministically until it visits an accepting state; then it can only move deterministically. The set \( Q \) can be divided into two disjoint parts \( Q_1 \) and \( Q_2 \) representing the states in the nondeterministic part and deterministic part, respectively; note that \( F \subseteq Q_2 \). The transition function \( \delta = \delta_1 \cup \delta_2 \cup \delta_3 \) (Section 5) then consists of the following three disjoint transition functions: \( \delta_1 : Q_1 \times \Sigma \rightarrow 2^{Q_1}, \delta_2 : Q_1 \times \Sigma \rightarrow 2^{Q_2}, \) and \( \delta_3 : Q_2 \times \Sigma \rightarrow 2^{Q_2} \), where the relation \( \delta_2 \) is deterministic. To simplify presentation, we impose the following two additional requirements on SDBAs: (1) we require that the entry points of \( Q_2 \) are accepting, i.e., \( \delta_1(q_1, a) \subseteq F \) for all \( a \in \Sigma \) and (2) we require that \( Q_1 \cap Q_2 \subseteq F \), i.e., all initial states in \( Q_2 \) are accepting. The two requirements guarantee that if a run in an SDBA touches a state in \( Q_2 \), it has already touched some accepting state. Any SDBA can be transformed into an equivalent SDBA satisfying the requirements by treating all non-accepting entry or initial states \( q \) from \( Q_2 \) (i.e., states from
either \((\bigcup_{a \in \Sigma} \delta_i(Q_1, a)) \setminus F\) or \((Q_1 \cap Q_2) \setminus F\) as follows: (1) we add a new accepting state \(q'\), (2) for all transitions entering \(q\) from \(Q_1\), we redirect them to \(q'\), and (3) we duplicate all outgoing transitions of \(q\) to \(q'\). Note that SDBAs recognize the same class of languages as BAs, but can be, in the worst case, exponentially larger.

3 Multi-Stage Generalization of Certified Modules

As mentioned in the introduction, a program \(P\) is represented by a BA \(A\). The termination proof of \(P\) can be obtained by decomposing \(A\) into several BAs \(A_1, \ldots, A_n\), whose languages jointly cover \(L(A)\), and then showing that each of them is terminating by means of a certified module [33].

Given a well-ordered set \((W, <)\), let \(\infty\) denote a value strictly larger than any other value in \(W\). In the following, we use \(\vec{v}\) to denote the vector of program variables of \(P\). A valuation \(\Phi\) is a function assigning a value to each variable from \(\vec{v}\). A statement is a command appearing in the program, such as an assignment or the guard of a while loop. The alphabet \(\Sigma\) is the set of statements appearing in \(P\). Each statement is associated with a binary relation over valuations representing the effect of the statement; for instance, the relation associated with the statement \(\Phi(i) \implies i > 0\). A Hoare triple is a triple \(\{\psi\} stmt \{\psi'\}\) where \(stmt\) is a statement and \(\psi, \psi'\) are predicates over program variables; a Hoare triple is valid if, for each pair of valuations \((\Phi, \Phi')\) in the relation associated with \(stmt\), if \(\Phi\) satisfies \(\psi\), then \(\Phi'\) satisfies \(\psi'\).

**Definition 3.1 (cf. [33, Definition 3])**. Given a BA \(A_M = (Q, \delta, \{q_1\}, \{q_f\})\) and a ranking function \(f_M\) from valuations into a well-ordered set \((W, <)\), we call a mapping \(I_M\) from states to predicates over program variables a rank certificate for \(f_M\) and \(A_M\) if the following properties hold:

- The initial state \(q_i\) is mapped by \(I_M\) to the predicate where the auxiliary variable oldrnk has the value \(\infty\), i.e., \(I_M(q_i) \iff \text{oldrnk} = \infty\).
- The accepting state \(q_f\) is mapped by \(I_M\) to a predicate in which the value of \(f\) over the program variables is strictly smaller than the value of the variable oldrnk, i.e., \(I_M(q_f) \iff f_M(\vec{v}) < \text{oldrnk}\).
- Each outgoing transition \(q \xrightarrow{stmt} q'\) from a state \(q \neq q_f\) corresponds to a valid Hoare triple, i.e., the triple \(\{I_M(q)\} stmt \{I_M(q')\}\) is valid.
- Each outgoing edge \(q_f \xrightarrow{stmt} q'\) from the accepting state \(q_f\) corresponds to a valid Hoare triple if we insert an additional assignment statement that updates oldrnk with the value of the ranking function, i.e., \(\{I_M(q_f)\} \iff \text{oldrnk} = f_M(\vec{v})\); stmt \(\{I_M(q')\}\) is valid.

We call \(M = (A_M, f_M, I_M)\) a certified module and define its language as \(L(M) = L(A_M)\). A certified module represents a set of paths in \(P\) that share the same termination argument.

That is, for all paths represented by \(M\), the evaluation of the ranking function \(f_M\) strictly decreases on visiting the accepting state \(q_f\).

3.1 The Multi-Stage Approach to Construct \(M\)

In this section, we describe our algorithm that generalizes an ultimately periodic word \(uv^\omega\) accompanied by a termination proof (obtained using an off-the-shelf termination checker) into a certified module (cf. Figure 1).

First, we construct the initial certified lasso module \(M_{uv^\omega}\) (cf. Section 3.1.1), which closely resembles the structure of \(uv^\omega\). The alphabet \(\Sigma\) of \(M_{uv^\omega}\) (and of its generalizations, see below) consists of all statements occurring in \(uv^\omega\). While such a module would work correctly in the later refinement, it is of a very limited practical use. In our experience, it usually covers only a very small fragment of program paths; sometimes it only covers the path corresponding to \(uv^\omega\).

The previous work [33] uses a generalization procedure that uses \(M_{uv^\omega}\) to construct a module \(M_{\text{nondet}}\) consisting of a nondeterministic BA (cf. Section 3.1.5). Although \(M_{\text{nondet}}\) is usually quite general, the drawback of this solution is the extremely high complexity of complementing a nondeterministic BA, which is performed in the subsequent step.

To alleviate this issue, we propose the following multi-stage approach for construction of certified modules.

Our multi-stage approach attempts to use the alphabet and states of \(M_{uv^\omega}\) to construct a certified module that is as easy to complement as possible, while also satisfying the condition that its language contains the word \(uv^\omega\) (so that when its language is removed from the set of uncertified traces, we are guaranteed to remove at least \(uv^\omega\)). The construction proceeds in stages, starting with a module that is the easiest to complement, and gradually progresses to modules whose complementation is harder (they exhibit a higher degree of nondeterminism), until it builds a module whose language contains \(uv^\omega\). As the last option, we construct \(M_{\text{nondet}}\), which is guaranteed to contain \(uv^\omega\) in its language, but is the hardest to complement.

In this work we consider 4 stages: besides stage 0, where the initial certified lasso module \(M_{uv^\omega}\) is built, we have at stage 1 the finite-trace certified module construction (cf. Section 3.1.2). The result contains a finite-trace BA whose complementation takes constant time; the generalization is, however, possible only under certain conditions. At stage 2, we build the deterministic certified module (cf. Section 3.1.3), which is relatively easy to complement since it is deterministic. If it does not suffice, at stage 3, we create the semideterministic certified module (cf. Section 3.1.4), which allows limited nondeterminism. The last construction we consider at stage 4 is the nondeterministic certified module \(M_{\text{nondet}}\) (cf. Section 3.1.5), which is guaranteed to accept \(uv^\omega\) but has the highest level of nondeterminism. More intermediate constructions can be added into this multi-stage approach.
### 3.1.1 Stage 0: Initial Certified Lasso Module

The initial certified lasso module $M_{\text{init}}$ consists of a BA accepting solely the word $uv^{\omega}$, a ranking function $f$, and a rank certificate $I$ (from a lasso program termination prover). The construction starts with the BA of the form depicted below, for the stem $u = u_1u_2 \ldots u_l$ and the loop $v = v_1v_2 \ldots v_m$.

![Diagram of BA and loop](image)

For instance, consider again the sorting program $p_{\text{sort}}$ and the $\omega$-word $uv^{\omega} = 12\doteq 0 \, j = 1 \, (j < i \, j++)^{\omega}$. The corresponding BA and rank certificate $I_{\text{sort}}$ are depicted below, where each state is annotated with the corresponding predicate given by $I_{\text{sort}}(i, j) = i - j$.

Module $M_{\text{init}}$ is obtained by generalizing the constructed BA by merging states with the same predicate. Note that for the given word $uv^{\omega}$, two states can be merged only if they both belong to the stem part $u$ or both belong to the loop part $v$; a state from the stem part can never be merged with a state from the loop part, since the former implies $\text{oldrnk} = \infty$ in its predicate while the latter implies its negation. If we merge such states for the BA from above, we obtain the following module accepting all words of the form $(i > 0) \, j = 1 \, (j < i \, j++)^{\omega}$, not just $uv^{\omega} = 12\doteq 0 \, j = 1 \, (j < i \, j++)^{\omega}$.

We denote the module from the example above as $M_{\text{init}}^{p_{\text{sort}}}$.

### 3.1.2 Stage 1: Finite-trace Certified Module

The first module we try to construct is the finite-trace certified module $M_1 = M_{\text{init}}$. This module can be constructed when $uv^{\omega}$ corresponds to a path that is infeasible already in the stem part. In such a case, there must be a state $q$ on some path from the initial state $q_i$ to the accepting state $q_f$ s.t. $I(q)$ is unsatisfiable and for every $q'$ on the shortest path from $q_i$ to $q$, $I(q')$ is satisfiable. More concretely, $M_{\text{init}}$ can be constructed from $M_{\text{init}}$ by (1) removing all states that are not on a path from $q_i$ to $q$, (2) removing all outgoing transitions of $q$, (3) adding self-loops $q \xrightarrow{\text{stmt}} q$ for all $\text{stmt} \in \Sigma$, and (4) setting $q$ as the single new accepting state.

\footnote{Note that if $u = \varepsilon$, the construction does not create a certified module (since $q_f = q_f$). As a remedy for this, in such a case we materialize $v$ once, i.e., we use the identity $e^{v^{\omega}} = v^{\omega}$, and continue in the standard way.}

### 3.1.3 Stage 2: Deterministic Certified Module

If $M_1$ cannot be constructed, we proceed by building a deterministic certified module $M_2 = M_{\text{det}}$. The high-level intuitive idea is to construct a DBA using sets of states of $M_{\text{init}}$ with transitions that respect the predicates of $M_{\text{init}}$ (so that the termination argument for $uv^{\omega}$ correctly extends to the whole language of the module) and are in some sense maximal. In particular, the successor of a set of states $\mathcal{Q}$ of $M_{\text{init}}$ over a statement $\text{stmt}$ is computed as the maximal set of states $\mathcal{Q}'$ satisfying the following property: the predicate of every state in $\mathcal{Q}'$ is a logical consequence of the conjunction of the predicates of the states in $\mathcal{Q}$ and the semantics of $\text{stmt}$.

For instance, let us consider an initial certified lasso module $M_{\text{init}}$ whose alphabet contains the statement $z = x + y$ and whose states $q_{23}, q_{42}, q_{65}$ are annotated by $I$ as follows: $I(q_{23})$ is $x = 23$, $I(q_{42})$ is $y = 42$, and $I(q_{65})$ is $z = 65$. Then, regardless of the transitions of $M_{\text{init}}$, the successor of $\{q_{23}, q_{42}\}$ over $z = x + y$ is the set $\{q_{23}, q_{42}, q_{65}\}$, since $I(q_{65})$ is implied by $I(q_{23}) \land I(q_{42})$ and the relation for $z = x + y$, which contains all pairs $(\Phi, \Phi')$ such that $\Phi(z) \implies z = x + y$.

**Definition 3.2.** Let $M_{\text{init}} = (\mathcal{A}, f, I)$ be an initial lasso module such that $\mathcal{A} = (Q, \delta, \{q_i\}, \{q_f\})$. The deterministic certified module $M_{\text{det}} = (\mathcal{A}_{\text{det}}, f, I_{\text{det}})$ with a DBA $\mathcal{A}_{\text{det}} = (Q_{\text{det}}, \delta_{\text{det}}, Q_{f_{\text{det}}}, I_{\text{det}})$ is defined as follows:

- The set of states of $\mathcal{A}_{\text{det}}$ is $Q_{\text{det}} = 2^Q$.
- Let $\delta_{\text{det}} : 2^Q \times \Sigma \rightarrow 2^Q$ be a function s.t. $\delta_{\text{det}}(\mathcal{Q}, \text{stmt}) = \{q' \in Q | \{q \in \mathcal{Q} | I(q)\} \text{stmt} \, I(q')\}$ is a valid Hoare triple, where $\text{stmt} = \{\text{oldrnk} := f(V)\}; \text{stmt} = q_f \in \mathcal{Q}$, otherwise $\text{stmt} = \text{stmt}$.
- Now, the transition function $\delta_{\text{det}}$ for a state $\mathcal{Q} \in Q_{\text{det}}$ and a statement $\text{stmt}$ is defined as $\delta_{\text{det}}(\mathcal{Q}, \text{stmt}) = \{\mathcal{Q}'\}$, where $\mathcal{Q}' = \delta_{\text{det}}(\mathcal{Q}, \text{stmt}) \quad \text{if} \quad q_f \notin \delta_{\text{det}}(\mathcal{Q}, \text{stmt})$, otherwise we omit all non-accepting states whose predicate contains $\text{oldrnk}$, i.e., $\mathcal{Q}' = \delta_{\text{det}}(\mathcal{Q}, \text{stmt}) \setminus \{q \in Q | q \neq q_f \land \text{oldrnk} \in \text{var}(I(q))\}$ where $\text{var}(I(q))$ denotes all variables occurring in $I(q)$.
- Note that the statement $\{\text{oldrnk} := f(V)\}$ is used only for defining $\delta_{\text{det}}$; it is not in the alphabet of $\mathcal{A}_{\text{det}}$.
- There is a single initial state, i.e., $Q_{f_{\text{det}}} = \{\{q_i\}\}$.
- The set of accepting states $I_{\text{det}}$ contains all states $\mathcal{Q} \in Q_{\text{det}}$ such that $q_f \in \mathcal{Q}$ or $\bigwedge_{q \in \mathcal{Q}} I(q)$ is unsatisfiable.

Moreover, $I_{\text{det}}$ is such that $I_{\text{det}} : \mathcal{Q} \mapsto \bigwedge_{q \in \mathcal{Q}} I(q)$.

By applying the certified deterministic module construction to $M_{\text{init}}^{p_{\text{sort}}}$, we obtain the following module:
Note that this module $M_{\text{sort}}^\text{det}$, despite accepting a non-empty language, is absolutely useless for the refinement of $M_{\text{sort}}$, since it rejects the word $uv^\omega = i > 0, j := 1 (j < 1, j++)^\omega$ representing the path whose termination has been proved.

While a DBA is easy to use in computing language difference, the certified deterministic module is not always useful, as we can see from the example above (in general, DBAs are known to be less expressive than BAs [3]).

### 3.1.4 Stage 3: Semideterministic Certified Module

In order to overcome the shortcomings of the deterministic certified module, we now present the semideterministic certified module $M_3 = M_{\text{semi}}$, which is $M_{\text{det}}$ enriched with additional transitions. In particular, for a statement $\text{stmt}$, each state $q$ that is not reachable from an accepting state $s.t. q_f \in \delta_\text{det}(Q, \text{stmt})$ (cf. Definition 3.2) has two $\text{stmt}$-successors:

- $\delta_\text{det}(Q, \text{stmt}) \setminus \{ q \in Q \mid q \neq q_f \land q \in \text{var}(I(q)) \}$, the original successor in $\delta_{\text{det}}$,

- an additional successor $\delta_\text{det}(Q, \text{stmt}) \setminus \{ q_f \}$.

Obviously, the resulting automaton is an SDBA that, furthermore, satisfies the requirements from Section 2 (the requirement that any run entering an accepting loop needs to enter via an accepting state—i.e., none of its states contains at the same time a state from the stem and a state from the loop— is guaranteed by the fact that all states in the stem imply oldrnk = $\infty$, while all states in the loop imply oldrnk < $\infty$).

By applying the certified semideterministic module construction to $M_{\text{sort}}^\text{det}$, we obtain the following module $M_{\text{semi}}^\text{sort}$.

Note that, in contrast to $M_{\text{sort}}^\text{det}$, the module $M_{\text{semi}}^\text{sort}$ already accepts the word $uv^\omega = i > 0, j := 1 (j < 1, j++)^\omega$.

### 3.1.5 Stage 4: Nondeterministic Certified Module

The nondeterministic certified module $M_4 = M_{\text{nondet}}$ is the most powerful generalization we considered. It is obtained by adding every possible transition between each pair of states to $M_{\text{sort}}^\omega$, as long as the rank certificate is still correct. For instance, the above lasso module $M_{\text{sort}}^\omega$ becomes the following nondeterministic module.

![Diagram of the nondeterministic module](image-url)

While usually accepting significantly more words than $M_{\text{sort}}^\omega$, the use of $M_{\text{nondet}}$ in the refinement can pose practical problems, caused by its high level of nondeterminism.

Although $M_{\text{nondet}}$ is always guaranteed to accept $uv^\omega$ (since it contains all transitions of $M_{\text{sort}}^\omega$), its use in the overall termination procedure is expensive, because algorithms for complementing BAs have a prohibitive complexity. Based on our experiments, constructing $M_{\text{nondet}}$ is seldom necessary, and in many cases, $M_{\text{semi}}$ is sufficient (in the worst case) for a successful generalization of a program path. As also observed in our experiments, computing the difference of a GBA representing program paths and a module can dominate the overall execution time, so constructing modules that are easier to complement is crucial. In the following sections, we provide efficient algorithms for computing the difference of a GBA and a BA (Section 4) and for complementing an SDBA (Sections 5 and 6) that serve as an enabling technology of the whole termination checking procedure.

### 4 Building Difference of a GBA and a BA

In this section, we introduce an algorithm that, given a GBA $A$ (in our setting representing program paths whose termination has not yet been established) and, in general, a BA $B$ (which represents the program paths whose termination we have just proved), constructs a GBA $D$ such that $L(D) = L(A) \setminus L(B)$. We present the algorithm and its optimizations in several steps. Note that we use GBAs since they are usually smaller than their equivalent BA counterparts and have a more efficient language intersection operation.

From a high-level view, our algorithm can be seen as an optimization of a naïve algorithm that first builds the complement of $B$, further denoted as $\overline{B}$, then constructs a GBA $A_I$ accepting the intersection $L(A_I) = L(A) \cap L(\overline{B})$ and, finally, removes useless states from $A_I$ (yielding an empty automaton in the case $L(A_I) = \emptyset$). Recall that a state $q$ is useless iff $L(A_I)(q) = \emptyset$, otherwise, $q$ is useful. Our optimizations that make the algorithm usable in practice are the following.

1. $\overline{B}$ is constructed on the fly when constructing $A_I$, i.e., only those states of $\overline{B}$ that occur in some product state of $A_I = A \cap \overline{B}$ are constructed (note that intersection of GBAs produces a GBA whose structure corresponds to finite automaton-like product construction).
Algorithm 1: Removing useless states from a GBA

Function remove_useless(𝒜):

1. Function construct(s):
2. Function is_nemp:
3. ++cnt; s.dfsnum ← cnt; is_nemp ← false;
4. SCCs.push((s, 𝒋(s))); act.push(s);
5. foreach q ∈ I do
6.  if q ̸∈ Q ∪ emp then // q ̸∈ Q ∪ [emp]
7.    construct(q);
8.  return 𝒜' = (Q', δ, Q', F') s.t. Vq ∈ Q : L(q) ≠ ∅
9.  Global: Q ← ∅, emp ← ∅, SCCs ← ∅, act ← ∅, cnt ← 0
10. if SCCs.push(1)
11.  while if
12.    if
13.      else
14.        act.pop();
15.      else if
16.        u = act.pop();
17.      else emp.add(u);
18.    while u ≠ s;
19. return is_nemp;

2. We remove useless states from 𝒜f using a modification of the state-of-the-art SCC-based algorithm for testing emptiness of the language of a GBA by Gaiser & Schwoon [26], which refines the algorithm of Couvreur [22] (Section 4.1).
3. When 𝒜 is an SDBA, we optimize the construction of 𝒜 from [12] by delaying nondeterministic choices as long as possible, thus significantly reducing the number of generated states (Section 5).
4. We prune the search from Point 2 by using an antichain-like [23] subsumption on the states of 𝒜f (Section 6).

4.1 Removing Useless States in a GBA

Algorithm 1 is a modification of the algorithm for checking emptiness of a GBA 𝒜 = (Q, δ, Qf, 𝒋) proposed by Gaiser & Schwoon [26] (GS for short), which is based on finding a reachable strongly connected component (SCC) that contains at least one state from every set 𝒇j.

Our modification not only tests the emptiness of 𝒥(𝒜), but also efficiently constructs a copy 𝒜' of 𝒜 without any useless state (and, therefore, if 𝒥(𝒜) = ∅, then 𝒜' is empty).

Similarly to GS, Algorithm 1 uses two stacks, SCCs and act, to keep track of the possible entry states of SCCs and the active states, which may be constructing the SCCs. Our algorithm uses additional data structures, namely the pair of sets Q' and emp, which are used to store all states that have been proved to be useful or useless. The algorithm starts in the function remove_useless and traverses the reachable states of 𝒜 in a depth first search manner. Each state has the data field dfsnum, which is used to record the relative order of the visit of the states, i.e., if t.dfsnum > s.dfsnum, then s has been visited before t. Therefore, if such a t can reach the said s and, at the same time, s is in act, this means that 𝒜 contains an SCC that includes both s and t. From all states forming an SCC, the one with the lowest value of dfsnum is the SCC’s entry point. The stack SCCs also assigns each possible SCC entry point q, the set of accepting conditions from F that q satisfies can infinitely many times reach (F(s) ⊆ {1, ..., k} denotes all accepting conditions that s belongs to).

The differences of Algorithm 1 from GS are the following:
1. If Algorithm 1 does not stop immediately when an accepting SCC is found (line 18), but continues in the construction,
2. In lines 25–26 (which correspond to leaving a possible SCC), the states popped from the stack act are classified to be either useful (then they are added to Q') or useless (then they are added to emp), and
3. We use Q' and emp in lines 10–11 to check whether we already know whether t has a non-empty language. The algorithm returns 𝒜 projected to the states Q', which are known to have non-empty languages.

Note that Algorithm 1 is amenable to on-the-fly traversal of the automaton 𝒜, i.e., 𝒜 can be provided implicitly.

Proposition 4.1. Algorithm 1 is correct.

5 Efficient Complementation of SDBAs

Algorithm 1 can be used for constructing the useful part of the GBA 𝒜 such that 𝒥(𝒜′) = 𝒥(𝒜) \ 𝒥(𝒜) and 𝒥(𝒜′) = 𝒥(𝒜), which requires an efficient construction of 𝒬. In this section, we present such a construction for an SDBA 𝒬.

We first explain the NCSB algorithm of Blahoudek et al. [12] for complementing SDBAs, which is, to the best of our knowledge, the most efficient complementation algorithm for SDBAs up to date. Later, we identify a source of inefficiency and propose a solution that mitigates it.

5.1 The NCSB-Original Algorithm

The NCSB-Original algorithm [12] can be viewed as an extension of the classical algorithm for complementing a nondeterministic finite automaton using the power set construction (used to determinize the automaton). The extension assigns
every state in a macro-state one of the labels \(\{N, C, S, C+B\}\) depending on the component where the state is present (as defined formally later, \(B\) is always a subset of \(C\) so the label \(C+B\) means that the state is both in \(C\) and \(B\)). The labels characterize the expected status of the runs going through the states. To avoid confusion, we will call a run of the complement automaton \(A_C = (Q_C, \delta_C, \Pi_C, I_C, F_C)\) a macro-run. We usually denote states in \(Q_C\) as \(\hat{q} = (N, C, S, B)\), where the components have the following intuitive meaning:

- **\(N\)** (nondeterministic): If a run of \(A\) touches a state in \(N\), then it is still in the nondeterministic part \(Q_1\).
- **\(C\)** (choice): If a run of \(A\) touches a state in \(C\), then it can never leave the set \(Q_2\), but we are not yet sure whether it is an accepting run. Therefore, every time a run in \(C\) leaves an accepting state, we nondeterministically guess whether it was the last time the run has touched an accepting state (in which case we move the run to the set \(S\)) or not (in which case it remains in \(C\)).
- **\(S\)** (safe): If a run arrives into \(S\), it can only remain safe, i.e., it will touch no more accepting states in the future. In the case the run is not safe, it will be blocked in \(A_C\) as soon as it attempts to touch an accepting state, i.e., if \(q \in S\) and \(\delta_C(q, a) = F\), then \(\delta_C(\hat{q}, a) = \emptyset\) (there can still be another safe run in some other guess though).
- **\(B\)** (breakpoint): The set \(B\) is used for tracking that all runs of \(A\) that arrive into \(Q_2\) will eventually become safe. In particular, once \(B\) becomes empty (denoting an accepting state), we copy the runs that are currently in \(C\) into \(B\), and remove them from \(B\) only when they become safe (i.e., when they have been moved to \(S\)).

The construction is formally defined as follows.

**Definition 5.1** (cf. [12, Section 3.2]). Given an SDBA \(A = (Q_1 \cup Q_2, \delta, \Pi, F)\), where \(Q_1\) and \(Q_2\) are defined as in Section 2, its complement automaton \(A_C = (Q_C, \delta_C, \Pi_C, I_C, F_C)\) is defined as follows:

- \(Q_C = \{(N, C, S, B) \in 2^C \times 2^C \times 2^O \times 2^F | B \subseteq C\}\).
- \(\Pi_C = \{(Q_1 \cap \Pi_1, Q_2 \cap \Pi_2, \emptyset, Q_2 \cap \Pi_1)\}\).
- \(F_C = \{ (N, C, S, B) \in Q_C | B = \emptyset \}\).
- \(\delta_C\) is the transition function \(\delta_C : Q_C \times \Sigma \rightarrow Q_C\) such that \((N', C', S', B') \in \delta_C((N, C, S, B), a)\) iff:
  1. \(N' = \delta_1(N, a)\).
  2. \(C' \cap S' = \delta_2(N, a) \cup \delta_3(C \cup S, a)\).
  3. \(C' \cap S' = \emptyset\).
  4. \(S' \supseteq \delta_5(S, a)\).
  5. \(C' \supseteq \delta_6(C \cap F, a)\), and
  6. \(B' = C'\) if \(B = \emptyset\), otherwise \(B' = \delta_8(B, a) \cap C'\).

Informally, rules 2–5 enforce that (1) the successors of states in \(S\) remain in \(S'\), (2) the successors of non-accepting states in \(C\) remain in \(C'\), (3) all accepting states in \(\delta_1(N, a) \cup \delta_3(C \cup S, a)\) stay in \(C'\), because \(S'\) is a set of non-accepting states, and (4) the rest of the states in \(\delta_2(N, a) \cup \delta_5(C \cup S, a)\) are nondeterministically partitioned into \(C'\) and \(S'\).

We note that the original definition [12] used yet another condition: "for all \(q \in C \setminus F\) it holds that \(\delta_5(q, a) \neq \emptyset\)." Since we assume the input BA to be complete (cf. Section 2), the condition always holds and hence we drop it. Also note that in order for the result of the NCSB algorithm to be complete, we may need to add a sink state (we hide this from the algorithm to make the presentation cleaner). When talking about the size of the set of states or transitions, we only consider those states and transitions reachable from \(\Pi_C\).

The best way to get an intuition about the algorithm is to simulate both accepting and rejecting runs of \(A\) in \(A_C\). Let \(\rho = q_0q_1 \ldots q_i \ldots\) be an accepting run of \(A\) over some word \(w \in \Sigma^*\) and \(q_i\) be the first accepting state in \(\rho\). Assume w.l.o.g. that \(q_0 \in Q_1\). It is easy to observe that for any macro-run \(\Pi = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1) \ldots (N_i, C_i, S_i, B_i) \ldots\), the run \(\rho\) is moved from \(N\) to \(C\) at position \(i\) (rule 2 and the fact that \(S\) is disjoint with \(F\)), i.e., \(q_k \in N_0\) for all \(1 \leq k \leq i - 1\) and \(q_i \in C_i\). (Moving a run from a set \(X\) to another set \(X'\) can be achieved by moving the corresponding state from \(X\) to \(X'\)). For any \(j > i\) with \(q_j \in F\), we have the following two cases (nondeterministic guessing by rules 2–5):

- **Case (1):** The run \(\rho\) is moved from \(C\) to \(S\) at a position \(j + 1\). In this case, \(\Pi\) will be blocked later at the position of the next occurrence of an accepting state in \(\rho\) (which there are infinitely many), because once \(\rho\) has moved to \(S\), it will stay in \(S\) (rule 4). It follows that \(\Pi\) is finite and, therefore, not an accepting macro-run.
- **Case (2):** If we assume that \(\rho\) stays in \(C\) for all such positions \(j\), then (rule 6) the run \(\rho\) will be copied to \(B\) the next time \(B\) becomes empty (if it ever happens). But then \(B\) cannot become empty again because \(\rho\) will stay inside it forever (our assumption is that \(\rho\) stays in \(C\) forever and hence also in \(B\) by rule 6). It follows that although \(\Pi\) is infinite, it is not an accepting macro-run.

On the other hand, we can show that if \(w \notin \mathcal{L}(A)\), we can construct from its rejecting runs \(\rho = q_0q_1 \ldots q_i \ldots\) an accepting macro-run \(\Pi = (N_0, C_0, S_0, B_0)(N_1, C_1, S_1, B_1) \ldots (N_i, C_i, S_i, B_i) \ldots (N_j, C_j, S_j, B_j) \ldots \) of \(A_C\). The strategy of the construction is simple. All such \(\rho\) will be moved from \(N\) to \(C\) at the first occurrence of an accepting state and then to \(S\) after the last occurrence of an accepting state. Because all states in \(C \cup S \cup B\) can only proceed via deterministic transitions from \(\delta_2\), there is only one corresponding run for each of them.

More concretely, we again have two cases: (i) if the run \(\rho\) never touches an accepting state, then \(\rho\) will stay in \(N\) forever (rule 1) and (ii) if \(q_i\) and \(q_j\) are the first and the last occurrence of an accepting state in \(\rho\) (it can happen that \(i = j\)), then there is a macro-run where \(\rho\) is moved from \(N\) to \(C\) at position \(i\) and then moved to \(S\) at position \(j + 1\) (rules 2–5). We can show that the following two conditions hold for such a macro-run \(\Pi\):

1. \(\Pi\) is non-blocking. From the definition of \(\delta_C\), a macro-state can become blocking only in one of two cases: (i) the
successor of a non-accepting state \( q \) in \( C \) coincides with the successor of some state in \( S \) (rules 3–5) or (2) the successor of a state in \( S \) is accepting (by definition of \( S \) and rule 4). The case (2) will never happen since \( \Pi \) moves a run to \( S \) only after the last occurrence of an accepting state. Suppose that the case (1) happens. Then, it means that there is no accepting state after \( q \) in the corresponding run, i.e., \( q \) is the successor of the last accepting state, which should already have been moved to \( S \), leading to a contradiction. So \( \Pi \) is non-blocking.

2. \( \Pi \) contains infinitely many accepting macro-states. Starting from any macro-state of \( \Pi \), no new runs can be moved to \( B \) until it becomes empty (rule 6). Since all runs \( \rho \) of \( \mathcal{A} \) on \( w \) are rejecting, they will be moved to \( S \) eventually after the last accepting state, i.e., no run can stay in \( B \) forever. The set \( B \) will eventually become empty and will be reset to \( C \) (rule 6). This will occur infinitely often.

### 5.2 Eager Guessing as the Source of Inefficiency

In this section, we show that complement automata constructed using NCSB-Original are unnecessarily large. Consider the example of an SDBA and its complement in Figure 3 (the figure shows only interesting parts of the automata).

Observe that in Figure 3b, the NCSB-Original algorithm made a guess at the macro-state \( (\emptyset, (q_1, q_2'), (q''_1), \emptyset) \). In fact, the construction needs to know whether \( \rho \) is in \( S \) or \( C \) only for the purpose of deciding whether a macro-state is accepting or rejecting (recall that \( B \subseteq C \)). In Figure 3b, we can find several macro-states (shown as \( \Box \)) that are redundant because the guessing of whether to keep a run \( \rho \) in \( C \) or move it to \( S \) was performed too eagerly.

A good point to do this guessing is to wait for \( B \) to become empty; before that we can simply keep the runs in \( C \cup S \) in the same set (in Figure 3b, we keep all of them in \( C \) in the leftmost branch of the complement automaton). If we do so, then none of the macro-states needs to be constructed. Note that their successors can be reconstructed from the macro-state \( (\emptyset, (q_1, q_2'), (q''_1), \emptyset) \). Having arrived at this macro-state, the guessing of all states in \( C \) has been postponed and hence any of them can nondeterministically either stay in \( C \) or be moved to \( S \) (dashed lines in Figure 3b).

To achieve the effect of delaying the guessing, our first attempt is to redefine the successor relation \( \delta'_C \) from Definition 5.1 such that \( (N', C', S', B') \in \delta'_C((N, C, S, B), a) \) iff

1. \( N' = \delta_1(N, a) \),
2. \( C' \cup S' = \delta_1(N, a) \cup \delta_2(C \cup S, a) \),
3. \( C' \cap S' = \emptyset \),
4. \[ \text{new} \] \( S' \supseteq \delta_2(S, a) \) if \( B = \emptyset \), otherwise \( S' = \delta_2(S, a) \), and
5. \[ \text{removed} \]
6. \( B' = C' \) if \( B = \emptyset \), otherwise \( B' = \delta_2(B, a) \cap C' \)

In particular, rule 4 has been exchanged for a new one and rule 5 has been removed. The new rule 4 enforces that all runs that are in \( S \) remain there and no new runs are added into \( S \) until an accepting macro-state (a macro-state where \( B = \emptyset \)) is encountered. Additionally, rule 5 from Definition 5.1 has been removed, so now one can nondeterministically move any non-accepting states from \( C \) to \( S \) when \( B \) becomes empty. The justification is that any run \( \rho \) in \( C \) must have had its guessing postponed (recall that if a run is in \( C \), it must have seen at least one accepting state) and in NCSB-Original could have been moved to \( S \) by now. A complement automaton constructed using NCSB-Original will have macro-runs corresponding to every postponed guessing, i.e., macro-runs traversing \( \Box \) macro-states in Figure 3b. Those macro-runs eventually reach successors of accepting macro-states produced by the modified algorithm, which in Figure 3b correspond to destinations of the dashed transitions.

Unfortunately, the change proposed above is not yet correct due to the issue that some run \( \rho \) in \( B \) has no chance to be moved to \( S \), even for the case when \( \rho \) has no accepting states after the state in \( B \). In such a case, \( \rho \) should correspond to an accepting run in the complement automaton \( \mathcal{A}_C \), but \( B \) can never become empty. This can be fixed by allowing the move of the successors of accepting states in \( B \) to \( S \) nondeterministically, i.e., guessing that it is the last occurrence of an accepting state in the run. This leads to an algorithm with lazy guessing, which we provide in the following section.

### 5.3 The NCSB-Lazy Algorithm

Combining the observations in the previous section, we obtain a new SDBA complementation algorithm, called NCSB-Lazy. The algorithm is obtained by redefining the transition function \( \delta'_C \) from Definition 5.1 such that \( (N', C', S', B') \in \delta'_C((N, C, S, B), a) \) iff

- When \( B = \emptyset \):
  a1. \( N' = \delta_1(N, a) \),
  a2. \( C' \cup S' = \delta_1(N, a) \cup \delta_2(C \cup S, a) \),
  a3. \( C' \cap S' = \emptyset \),
  a4. \( S' \supseteq \delta_2(S, a) \), and
  a5. \[ \text{removed} \]
  a6. \( B' = C' \).
- When \( B \neq \emptyset \):
  b1. \( N' = \delta_1(N, a) \),
  b2. \[ \text{new} \] \( B' \cup S' = \delta_2(B \cup S, a) \),
  b3. \[ \text{new} \] \( B' \cap S' = \emptyset \),
  b4. \( S' \supseteq \delta_2(S, a) \),
  b5. \[ \text{new} \] \( C' = (\delta_2(C, a) \cup \delta_1(N, a)) \setminus S' \), and
  b6. \[ \text{new} \] \( B' \supseteq \delta_2(B \setminus F, a) \).

When \( B = \emptyset \), the construction works in the same way as the one presented in Section 5.2. When \( B \neq \emptyset \), rules b2–b4 and b6 enforce that (1) a run in \( B \) that touches an accepting state can be nondeterministically moved to \( S \) and (2) a run in \( S \) will remain in \( S \) forever. Rule b5 enforces that if a state is moved from \( B \) to \( S \), then it should also be removed from \( C \).

**Proposition 5.2.** The complement \( BA \) constructed by NCSB-Lazy contains at most as many macro-states as the complement \( BA \) constructed by NCSB-Original.
Below we give a lemma that will be used in the correctness proof of NCSB-Lazy.

**Lemma 5.3.** Consider an SDBA $\mathcal{A}$, its complement $\mathcal{A}_C$ constructed by NCSB-Lazy, a word $w \in \Sigma^\omega$, and a macro-state $\hat{\rho} = (N, C, S, B)$ from $\mathcal{A}_C$. Further, assume that for all runs $\rho$ over $w$ in $\mathcal{A}$, it holds that

1. if $\rho$ starts from a state $q \in N \cup C \cup S$, it is rejecting,
2. if $\rho$ starts from a state $q \in S$, it is safe, and
3. if $\rho$ starts from a state $q \in B$, it is not safe.

Then one can construct an accepting macro-run over $w$ in $\mathcal{A}_C$.

**Proof.** Our strategy for constructing an accepting macro-run $\Pi$ from $\hat{\rho}$ is the following. If $B = \emptyset$, we move all safe runs in $C$ into $S$ and copy all unsafe runs to the $B$ component of the next macro-state. If $B \neq \emptyset$, we move all runs in $B$ into $S$ immediately when they become safe, i.e., immediately after they touch an accepting state for the last time. The other parts of the construction of $\Pi$ are deterministic, i.e., one can construct deterministically every macro-state in $\Pi$ following the transition relation of NCSB-Lazy. We now show that $\Pi$ is an accepting macro-run in $\mathcal{A}_C$ by proving two properties.

1. $\Pi$ is non-blocking. From the definition of the transition relation of NCSB-Lazy, a macro-state can become blocking only in one of the following cases: (1) the successor of a non-accepting state $q$ in $B$ coincides with the successor of some state in $S$ (rules b3, b4, and b6) and (2) the successor of a state in $S$ is accepting (by definition of $S$ and rule b4). The case (2) will never happen since, as defined above, $\Pi$ moves a run to $S$ only after the last occurrence of an accepting state. Suppose that case (1) happens. Then, it means that there is no accepting state after $q$ in the corresponding run, i.e., $q$ is the successor of the last accepting state, which should have already been moved to $S$, leading to a contradiction.

2. $\Pi$ contains infinitely many accepting macro-states. Starting from any macro-state of $\Pi$, no new runs can be moved to $B$ until it becomes empty (rule a6). Since all runs $\rho$ of $\mathcal{A}$ are rejecting starting from any states in $\hat{\rho}$, they will be moved to $S$ eventually after the last accepting state, i.e., no run can stay in $B$ forever. The set $B$ will eventually become empty and will (infinitely often) be reset to $C$ (rule a6). □

**Theorem 5.4.** Given an SDBA $\mathcal{A}$, NCSB-Lazy produces a BA $\mathcal{A}_C$ such that $L(\mathcal{A}_C) = \Sigma^\omega \setminus L(\mathcal{A})$.

**Proof.** The case that $w \in L(\mathcal{A})$ implies $w \notin L(\mathcal{A}_C)$ can be proved in a similar way as in NCSB-Original. In particular, we need to show that any accepting run $\rho$ of $\mathcal{A}$ over $w$ will either stay forever in $B$ or move to $S$ and block the macro-run. For the case that $w \notin L(\mathcal{A})$ implies $w \in L(\mathcal{A}_C)$, we can construct an accepting macro-run $\Pi$ from the runs $\rho$ of $\mathcal{A}$ on $w$ using Lemma 5.3. In order to do so, we need to ensure that the initial macro-state $(Q_1 \cap Q_2 \cap Q_3, 0, Q_2 \cap Q_1)$ satisfies the requirements of Lemma 5.3, i.e., for all runs $\rho$ over $w$ in $\mathcal{A}$, it holds that

1. if $\rho$ starts from a state $q \in N \cup C \cup S$, it is rejecting,
2. if $\rho$ starts from a state $q \in S$, it is safe, and
3. if $\rho$ starts from a state $q \in B$, it is not safe.

Requirement 1 is satisfied because $w$ has only rejecting runs from the initial states $Q_1$. Requirement 2 is satisfied because $S = \emptyset$. Requirement 3 is satisfied because all states in $Q_2 \cap Q_1$ are also in $F$ (due to our restriction on SDBAs, cf. Section 2), so runs starting from them are not safe. □

From the experimental results (cf. Section 7), one can see that although the changes in the algorithm are small, they induce a large difference in performance. We believe that the idea of delaying nondeterministic choices can be useful in other algorithms, such as rank-based BA complementation.
6 Subsumption-based Pruning in the Construction of a Difference Automaton

In this section we describe subsumption relations that can be used to optimize the construction of the difference automaton described in Section 4. The subsumption relations are, in a way, similar to the so-called antichain \cite{2,23} algorithms used in language inclusion and universality testing over nondeterministic finite automata.

We start by describing the notation used in this section. For macro-states \(\hat{p} = (N_p, C_p, S_p, B_p)\) and \(\hat{r} = (N_r, C_r, S_r, B_r)\), we define the following two subsumption relations:

\[
\hat{p} \ll\hat{r} \quad \text{def} \quad N_p \supseteq N_r \land C_p \supseteq C_r \land S_p \supseteq S_r \quad \text{and (4)} \\
\hat{p} \ll\hat{r} \quad \text{def} \quad \hat{p} \subseteq \hat{r} \land B_p \supseteq B_r. \quad \text{(5)}
\]

Let \(A^O_r\) and \(A^L_r\) be the complement automata constructed using NCSB-Original and NCSB-Lazy, respectively. We define two language inclusion relations \(\subseteq^O_r\) and \(\subseteq^L_r\) over macro-states \(\hat{p}\) and \(\hat{r}\) as follows:

\[
\hat{p} \subseteq^O_r \hat{r} \quad \text{def} \quad L_{A^O_r}(\hat{p}) \subseteq L_{A^O_r}(\hat{r}), \quad \text{(6)} \\
\hat{p} \subseteq^L_r \hat{r} \quad \text{def} \quad L_{A^L_r}(\hat{p}) \subseteq L_{A^L_r}(\hat{r}). \quad \text{(7)}
\]

As a consequence, we can use \(\subseteq\) and \(\subseteq^B\) in Algorithm 1 (when computing the difference automaton \(A \setminus B\)) for early termination when checking whether a language of an encountered macro-state is empty (lines 3 and 11). In particular, we change testing (non-)membership of a macro-state \(\hat{q}\) in the set \(\text{emp}\) into testing the same in the set \(\text{emp}\) defined as

\[
[\text{emp}] = \{ (q, \overline{q'}_B) \mid \exists (q, \overline{q'}_B) \in \text{emp} : \overline{q'}_B \preceq e \overline{q'}_B \}, \quad \text{(10)}
\]

where \(\preceq \in (\subseteq, \subseteq^B)\) depending on the particular algorithm used for complementation (\(\subseteq\) for NCSB-Original and \(\subseteq^B\) for NCSB-Lazy). Note that on line 26, \(\text{emp}\) can be maintained in the form of an antichain, i.e., to contain only elements incomparable w.r.t. \(\preceq\).

6.1 Subsumption Relation for NCSB-Original

We first show that \(\hat{p} \ll\hat{r} \implies \hat{p} \ll\hat{r}\) for any two macro-states \(\hat{p} = (N_p, C_p, S_p, B_p)\) and \(\hat{r} = (N_r, C_r, S_r, B_r)\). We prove this fact by constructing a strategy that for any accepting macro-run from \(\hat{p}\) returns an accepting macro-run from \(\hat{r}\) over the same word. Our proof consists of two parts. First, we define two new notions of simulation relation, named early simulations, between traces and states of a BA and we show that they under-approximate language inclusion. Second, we prove that both subsumption relations \(\subseteq\) and \(\subseteq^B\) are instances of the corresponding early simulation relations.

6.1.1 Early Simulation

Consider a BA \(A = (Q, \delta, Q_I, F)\) and a pair of traces \(\pi_p = p_0 \overrightarrow{w_0} p_1 \overrightarrow{w_1} \cdots\) and \(\pi_r = r_0 \overrightarrow{w_0} r_1 \overrightarrow{w_1} \cdots\) over the word \(w = w_0w_1 \cdots \in \Sigma^\omega\) from the states \(p_0 \in Q\) and \(r_0 \in Q\). We say that \(\pi_p\) is early simulated by \(\pi_r\) (or, alternatively, that \(\pi_r\) early simulates \(\pi_p\)), denoted as \(\pi_p \preceq \pi_r\), iff

\[
\forall i < j : ((p_i \in F \lor i = -1) \land p_j \in F) \implies \exists i < k < j : r_k \in F, \quad \text{(11)}
\]

and that \(\pi_p\) is early+1 simulated by \(\pi_r\) (written \(\pi_p \preceq_{e^1} \pi_r\)) iff

\[
\forall i < j : (p_i, p_j) \in F \implies \exists i < k < j : r_k \in F. \quad \text{(12)}
\]

Intuitively, the early+1 simulation requires that between every two \(\pi_p\) touches an accepting state, \(\pi_r\) also touches an accepting state; the early simulation further requires that \(\pi_r\) first touches an accepting state not later than \(\pi_p\) does.

We extend the proposed notions of simulation to states as follows. First, we define a strategy as a function \(\sigma : Q \times (Q \times \Sigma \times Q) \rightarrow (Q \times \Sigma \times Q)\) such that \(\sigma(r, p \overrightarrow{a} p') = r \overrightarrow{a} r'\) where \(r' \in (r, a)\). That is, \(\sigma\) picks a transition from \(r\) based on the transition \(p \overrightarrow{a} p'\) selected by the environment. Next, we lift strategy to traces such that for a trace \(\pi_p\) defined as above, we set \(\sigma(r_0, \pi_p) = r_0 \overrightarrow{w_0} r_1 \overrightarrow{w_1} \cdots\) where for all \(i \geq 0\) it holds that \(\sigma(r_i, \overrightarrow{w_0} r_{i+1}) = r_i \overrightarrow{w_i} r_{i+1}\). We say that \(\pi_0\) is early (resp. early+1) simulated by \(\pi_0\), denoted as \(\pi_0 \preceq \pi_0\) (resp. \(\pi_0 \preceq_{e^1} \pi_0\)) iff there exists a strategy \(\sigma_{\pi_0}\) (resp. \(\sigma_{\pi_0}^{e^1}\)) such that for every trace \(\pi_0\) starting in \(\pi_0\), it holds that \(\pi_0 \preceq \pi_{\sigma(\pi_0)}\) (resp. \(\pi_0 \preceq_{e^1} \pi_{\sigma^{e^1}(\pi_0, \pi_0)}\)).

The following proposition states that the introduced simulations under-approximate language inclusion.

**Proposition 6.1.** Given a BA \(A\), the following holds for the relations over the states of \(A\):

\[
\subseteq_e \subseteq \subseteq_{e^1} \subseteq \subseteq. \quad \text{(13)}
\]

6.1.2 The Subsumption \(\subseteq\) is an Early Simulation

Consider an SDBA \(A\) and its complement BA \(A^O\) constructed by NCSB-Original, and let us fix the following two states of \(A^O: \hat{p} = (N_p, C_p, S_p, B_p)\) and \(\hat{r} = (N_r, C_r, S_r, B_r)\).

**Lemma 6.2.** The relations \(\subseteq\) and \(\subseteq^B\) on \(A^O\) are an early+1 simulation and an early simulation respectively:

\[
\hat{p} \subseteq \hat{r} \implies \hat{p} \subseteq_{e^1} \hat{r} \quad \text{and}\quad \hat{p} \subseteq^B \hat{r} \implies \hat{p} \subseteq \hat{r}. \quad \text{(14)}
\]

Proof of (14). We use the strategy \(\sigma_{\hat{p}}\) that for a transition \(\hat{p} \overrightarrow{a} \hat{p}' = (N_p, C_p, S_p, B_p)\) chooses a transition \(\hat{r} \overrightarrow{a} \hat{r}' = (N_r, C_r, S_r, B_r)\) that respects all nondeterministic choices made in \(\hat{p} \overrightarrow{a} \hat{p}'\). In particular, if the successor \(q'\) of a state \(q \in C_p \cap C_r\) was moved to \(S_r\), i.e., \(q' = \delta(q, a) \in S_p\), the strategy \(\sigma_{\hat{p}}\) will also move \(q'\) to \(S_r\), otherwise \(q'\) will stay in \(C_r\). Other parts of the construction of \(\hat{r}'\) are deterministic.
(i.e., just follow the definition of \( \delta_C \) in Definition 5.1). The strategy guarantees that (1) \( \widetilde{p} \sqsubseteq \widetilde{r} \implies \widetilde{p}' \sqsubseteq \widetilde{r}' \) and (2) the transition \( \widetilde{r} \xRightarrow{\pi} \widetilde{r}' \) exists (proof omitted due to lack of space).

Next, we show that for any two traces \( \pi_p = \widetilde{p}_0 \xrightarrow{w_0} \widetilde{p}_1 \xrightarrow{w_1} \cdots \) and \( \pi_r = \sigma_C(r_0, \pi_p) = \widetilde{r}_0 \xrightarrow{w_0} \widetilde{r}_1 \xrightarrow{w_1} \cdots \) such that \( \widetilde{p}_0 = \widetilde{p} \) and \( \widetilde{r}_0 = \widetilde{r} \), the condition \( \pi_p \preceq e_{\varepsilon} \pi_r \) is satisfied, i.e., \( \forall i < j : (\widetilde{p}_i, \widetilde{p}_j) \in F_\pi \) implies \( \exists i < k < j : \widetilde{r}_k \in F_\pi \), where \( F_\pi \) is the set of accepting macro-states of \( \mathcal{A}^O_C \).

**Claim:** For all \( i \geq 0 \), if \( \widetilde{p}_i \in F_\pi \), then \( \pi_p \preceq e_{\varepsilon} \pi_r \).

**Proof:** Let \( \mathcal{A}^O_C \) be a BA produced by NCSB-Lazy from \( A \) and \( \pi_p \) and \( \pi_r \) be a pair of macro-states of \( \mathcal{A}^O_C \). In general, \( \pi_p \subseteq \pi_r \) does not imply \( \pi_p \preceq \pi_r \). In particular, let \( q \) be a non-accepting state of \( A \) with only one outgoing transition that is a self-loop and let \( \pi_p = (0, \{q\}, 0, 0) \) and \( \pi_r = (0, \{q\}, 0, \{q\}) \). Note that \( \pi_p \subseteq \pi_r \) (as \( \pi_p \) does not relate the \( B \) components). Also note that there exists an accepting macro-run \( \pi_p : (0, \{q\}, 0, \{q\}) \omega \) from \( \pi_p \), while there exists no accepting macro-run from \( \pi_r \). Since the \( B \) component of \( \pi_r \) can never become empty (cf. condition b in Section 5.3).

Fortunately, the stronger subsumption relation \( \preceq_B \) still under-approximates the language inclusion \( \preceq_L \), so it can be used to optimize the difference automaton construction.

**Theorem 6.4.** The relation \( \preceq_B \) under-approximates language inclusion of macro-states in a complement automaton constructed using NCSB-Lazy:

\[
\preceq_B \subseteq \preceq_L. \tag{19}
\]

**Proof.** Let \( \pi_p = (N, C, S, B) \) and \( \pi_r = (N', C', S', B') \) s.t. \( \pi_p \preceq_B \pi_r \). For any word \( w \in L_{\mathcal{A}_C^O}(\pi_p) \), it is not difficult to observe that:

- all runs over \( w \) from states in \( N \cup C \cup S \) are rejecting and the runs from states in \( N' \cup C' \cup S' \) are their subset,
- all runs over \( w \) from states in \( S \) are safe, and the runs from states in \( S' \) are their subset, and
- all runs over \( w \) from states in \( B \) are unsafe, and the runs from states in \( B' \) are their subset.

Hence, we can apply Lemma 5.3 to obtain an accepting macro-run from \( \pi_r \) over \( w \) in \( \mathcal{A}^L_C \). It follows that \( \pi_p \preceq_L \pi_r \).

## 7 Experimental Evaluation

We implemented the presented techniques as an extension of Ultimate Automizer [33] and experimentally evaluated their performance. The results are presented in Figures 4 and 5. The points in the top/right-most regions of the plots represent experiments where the corresponding setting timed out (>300 s) or went out of available memory (4 GiB).

In the first set of experiments, shown in Figure 4, we evaluated the performance of three versions of the SDBA complementation algorithm: NCSB-Original from [12], NCSB-Lazy (Section 5.3), and NCSB-Lazy with subsumption (Section 6.2). We used the set of all 1159 SDBAs produced by Ultimate Automizer during termination analysis of all non-recursive benchmarks (1575 programs) in the Termination category of SV-Comp [1, 10].

\footnotetext[2]{We used 6 different settings and collected all SDBAs produced before the timeout. Because Ultimate Automizer constructs the difference of...}
Figure 4. Comparing the performance of NCSB-Lazy with NCSB-Original and evaluating the effect of subsumption.

Figure 5. Evaluating the performance of the multi-stage approach and the optimized difference operation [s]

In the next experiment, we evaluated the performance of the proposed optimizations within program termination analysis. We again use all non-recursive programs from the Termination category of SV-Comp. In the left-hand side of Figure 5, we evaluated the performance of the multi-stage approach w.r.t. the single-stage approach (which always directly generalizes a counterexample to a nondeterministic module $M_{nondet}$). For the multi-stage approach, we first evaluated three different generalization sequences:

(i) $M_{untw} \rightarrow M_{fin} \rightarrow M_{semi} \rightarrow M_{nondet}$ (we skip $M_{det}$)
(ii) $M_{untw} \rightarrow M_{fin} \rightarrow M_{det} \rightarrow M_{nondet}$ (we skip $M_{semi}$)
(iii) $M_{untw} \rightarrow M_{fin} \rightarrow M_{det} \rightarrow M_{semi} \rightarrow M_{nondet}$

All of them solved roughly the same amount of examples ($\pm 2$ in the set of 1375 programs) when the SDBA difference optimization was not used. Therefore, we chose option (i), which produces the most SDBAs, so we can exploit the full potential of our optimizations. Using this option, the analysis of 1375 programs generated 6375 finite-trace modules, 1200 semideterministic modules, and 3 nondeterministic modules. We can see that the multi-stage approach solves significantly more cases than the single-stage approach (fewer points in the up-most region of the plot). The improvement is obtained mainly by avoiding the construction of $M_{nondet}$, which has a costly complementation procedure. The occasional slowdown can still happen since different counterexample generalization constructions produce BAs with different languages (in the subsequent steps, we then obtain different counterexamples, giving rise to a different global search space).

In the right-hand side of Figure 5, we evaluated the performance of the proposed optimizations of the difference automaton construction ("Multi-stage + opt" uses NCSB-Lazy + subsumption to complement SDBAs). We can observe that there are some cases where the version with optimizations has a worse time or even times out, but the version without optimizations can solve them. This can happen because of one of the following reasons: (1) the subsumption techniques impose overhead on the execution time or (2) NCSB-Lazy produces more transitions than NCSB-Original. We can, however, clearly see that the proposed optimizations are indeed helpful in the overall performance of the termination analysis, as the number of solved cases is significantly higher than for the version without them.
In particular, the number of benchmarks that timeouted or ran out of memory is for the various settings as follows:

- Single-stage: 691
- Multi-stage without optimizations: 296
- Multi-stage with Subsumption: 253
- Multi-stage with NCSB-Lazy: 250
- Multi-stage with NCSB-Lazy + Subsumption: 249

One can see that both NCSB-Lazy and Subsumption are already quite useful to improve the overall performance, but the best result is obtained by turning all optimizations on.

8 Related Work

To the best of our knowledge, there is no other tool implementing a termination analysis closely related to the algorithm implemented in Ultimate Automizer. Hence, our evaluation in Section 7 was focused on different variations of this algorithm. For a comparison with other tools, we rather refer to the results of the independent competition on software verification SV-Comp. In SV-Comp 2018 [10] the Ultimate Automizer team used the optimizations that were presented in this paper and won the Termination category.\(^3\)

For the sake of completeness, we give a brief overview of other termination analyses that make use of automata or that have participated in SV-Comp.

One line of research is based on the \textit{size-change principle} [4, 5, 37]. In this technique, one examines the flow of values among variables in each code block, which values are bounded from below (e.g., by the condition of an if statement), which values are not increasing, and which values are decreasing. The basic idea of this approach is that if on each (infinite) path there is at least one value that is bounded from below, never increasing, and infinitely often decreasing, then the program terminates. This property is inferred using one of two techniques. One is based on BAs, the other is based on Ramsey’s theorem. In contrast to our approach, there is only one BA, in general not semideterministic. The BA is, however, reverse-deterministic, which also allows a more efficient language inclusion check [24]. Although this approach and the approach implemented in Ultimate Automizer both use BAs, they are not closely connected. Using size-change termination, one always has a fixed domain of constraints, which is used to track decreasing values, whereas in Ultimate Automizer we may use ranking functions to track the values of arbitrary expressions. Furthermore, Ultimate Automizer infers the ranking functions on-demand and splits the program into components where these ranking functions serve as termination proofs.

Another line of research is based on \textit{transition invariants} [45]. There, the basic idea is that if the transitive closure of the program’s transition relation is a subset of a union of well-founded relations, then the program is terminating.

In this line of research, soundness is also proved via Ramsey’s theorem. The approach has been first implemented in the Terminator tool [17], and later also in T2 [15] and CPAchecker [11, 41]. The set of well-founded relations used there is obtained from a set of functions, where each of them is a ranking function for a set of paths in the program. Terminator constructs this set of functions incrementally in the following CEGAR-style algorithm. The tool lets a safety checker analyze if, in each loop location, at least one of the functions is a ranking function. If not, a lasso-shaped counterexample is obtained and its termination is analyzed by techniques specialized for lasso-shaped programs [6–9, 14, 16, 32, 39, 44]. If the counterexample is spurious, i.e., the lasso-shaped program is terminating, the ranking function of this path is constructed and added to the set of functions. This process is repeated until a real counterexample is found or the safety checker detected that, for each path, one of the functions is a ranking function. A bottleneck of this approach is that the safety checks become costlier over time since the set of ranking functions is growing. In Ultimate Automizer, this bottleneck is shifted from a program analysis task to an automata theory task. We never have to combine several ranking functions since the program is decomposed into several modules and there is only one function for each module. This comes at the price that the number of modules is growing over time, so the automata operations that are applied to these modules also become costlier.

The AProVe tool [28] first applies several transformations (e.g., removing pointers [48]) to translate a program into an integer term rewriting system. Afterwards, it applies various techniques to analyze termination of the resulting system [25, 29]. Termination can also be analyzed via an abstract interpretation framework [21]. Several abstract domains have been developed [20, 49, 52, 53] and implemented in the FuncTion tool [50]. In contrast to Ultimate Automizer, which is decomposing the set of program traces, there are also tools that decompose the state space of the program, such as HipTNT+ [36] and SEAHORN [51]. Decomposing the state space allows them to infer ranking functions for each component separately.

Acknowledgments

We thank the reviewers for their suggestions, which significantly helped to improve the readability of the paper. The work on this paper was supported by the Czech Science Foundation project 17-12465S, the IT4iXS: IT4Innovations Excellence in Science project (LQ1602), the FIT BUT project FIT-S-17-4014, the National Natural Science Foundation of China (Grants No. 61532019, 61650410658, 61761136011), the CAS/SSAFEA International Partnership Program for Creative Research Teams, the CDZ project CAP (GZ 1023), and the Ministry of Science and Technology of Taiwan (project 106-2221-E-001-009-MY3).

\(^3\)https://sv-comp.sosy-lab.org/2018/results/results-verified/
References


