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.

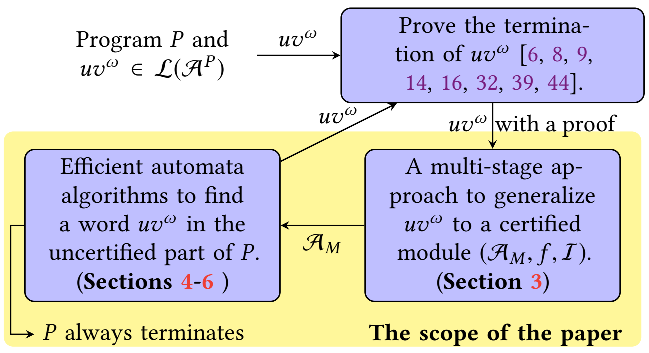
CCS Concepts • **Theory of computation → Automata over infinite objects**; • **Software and its engineering → Formal software verification**;

Keywords Program Termination, Büchi Automata Complementation and Language Difference

ACM Reference Format:
Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced Automata-Based Algorithms for Program Termination Checking. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192405

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



Program P and $uv^\omega \in \mathcal{L}(\mathcal{A}^P)$
uv^ω
Prove the termination of uv^ω [6, 8, 9, 14, 16, 32, 39, 44].
uv^ω
uv^ω with a proof
Efficient automata algorithms to find a word uv^ω in the uncertified part of P. (Sections 4-6)
$\mathcal{A}_M$
A multi-stage approach to generalize uv^ω to a certified module $(\mathcal{A}_M, f, \mathcal{I})$. (Section 3)
P always terminates
The scope of the paper

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

devise a termination argument that holds for any possible interleaving of different paths through the loop body?

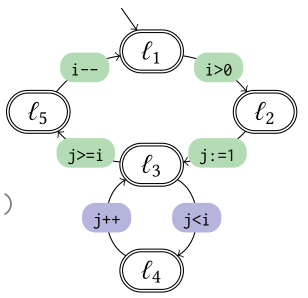
Due to the difficulty of solving the general problem, many researchers have focused on its simplified version that addresses only *lasso-shaped* programs, i.e., programs where the control flow consists of a stem followed by a simple loop without any branching. Proving termination of this class of programs 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 $\tau = uv^\omega$ from the possible behaviors of P and attempts to prove its termination using an off-the-shelf approach. Second, it generalizes τ into a (potentially infinite) set of paths $\mathcal{M}$, called a *certified module*, that all share the same termination proof with τ. Finally, it checks whether the behavior of P contains a path τ' 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^{sort} in Figure 2a. Figure 2b shows the control flow graph (CFG) of P^{sort} as a Büchi automaton (BA) $\mathcal{A}^{P^{\mathsf{sort}}}$. The alphabet of $\mathcal{A}^{P^{\mathsf{sort}}}$ is the set of all statements occurring in P^{sort} and each state of $\mathcal{A}^{P^{\mathsf{sort}}}$ is a location of P^{sort}. All states of $\mathcal{A}^{P^{\mathsf{sort}}}$ are accepting so every infinite sequence of statements of the program corresponds to an infinite word in the language $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}})$. The task is to decompose $\mathcal{A}^{P^{\mathsf{sort}}}$ into a finite set of BAs $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, each representing a program with a termination argument, such that $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \cdots \cup \mathcal{L}(\mathcal{A}_n)$, so every path of P^{sort} is represented by a word in $\mathcal{A}^{P^{\mathsf{sort}}}$ 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* $\mathcal{I}_i$ mapping each state to a *predicate* over the program variables (cf. Section 3). The triple $\mathcal{M}_i = (\mathcal{A}_i, f_i, \mathcal{I}_i)$ is called

```
program sort(int i):
ℓ1: while (i>0):
ℓ2:   int j:=1
ℓ3:   while (j<i):
        // if (a[j]>a[i]):
        //   swap(a[j],a[i])
ℓ4:     j++
ℓ5:   i--
```

(a) Program P^{sort}



(b) The BA $\mathcal{A}^{P^{\mathsf{sort}}}$

Figure 2. An example program and its BA representation

a *certified module*. The construction of the set $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ is step-wise (see Figure 1). First, we find an ultimately periodic word $uv^\omega \in \mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}})$—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 $uv^\omega =$ `i>0` `j:=1` (`j<i` `j++`)$^\omega$. We can prove termination of the path corresponding to uv^ω by finding, e.g., the ranking function $f_1(\mathtt{i}, \mathtt{j}) = \mathtt{i} - \mathtt{j}$.

In the following, we will denote the outer loop of $\mathcal{A}^{P^{\mathsf{sort}}}$ as $\textsc{Outer} =$ `j>=i` `i--` `i>0` `j:=1` and its inner loop as $\textsc{Inner} =$ `j<i` `j++`. We can observe that f_1 is also a ranking function for the set of paths obtained by generalizing uv^ω into the set of words that correspond to all paths that eventually stay in the inner loop, i.e., words from

$$\mathcal{L}_1 = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner} + \textsc{Outer})^* \cdot \textsc{Inner}^\omega. \tag{1}$$

The language $\mathcal{L}_1$ together with a ranking function f_1 and a rank certificate $\mathcal{I}_1$ can be represented by the certified module $\mathcal{M}_1 = (\mathcal{A}_1, f_1, \mathcal{I}_1)$ where $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}_1$. We proceed by removing all paths covered by $\mathcal{L}_1$ from $\mathcal{A}^{P^{\mathsf{sort}}}$ 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 uv^ω on failure). In our example, the difference corresponds to the (non-empty) language

$$\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1}) = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner}^*\textsc{Outer})^\omega \tag{2}$$

represented by $\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1}$. Suppose the next sampling gives us $uv^\omega =$ `i>0` `j:=1` $\cdot\ \textsc{Outer}^\omega$, for which, e.g., the ranking function $f_2(\mathtt{i}, \mathtt{j}) = \mathtt{i}$ 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

$$\mathcal{L}_2 = \texttt{i>0}\ \texttt{j:=1} \cdot (\textsc{Inner}^*\textsc{Outer})^\omega. \tag{3}$$

We represent these paths by the certified module $\mathcal{M}_2 = (\mathcal{A}_2, f_2, \mathcal{I}_2)$ where $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}_2$. After removing the words from $\mathcal{L}(\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1})$, we, finally, obtain the BA $\mathcal{A}^{P^{\mathsf{sort}}}_{|\mathcal{A}_1, \mathcal{A}_2}$, whose language is empty. This means that the modules $\mathcal{M}_1$ and $\mathcal{M}_2$ cover all possible paths of the program P^{sort} and, because each of them comes with a termination argument, we can conclude that all paths of P^{sort} are guaranteed to terminate.

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 uv^ω, we consider the following subclasses: (i) *finite-trace BAs* (BAs accepting the language $w\Sigma^\omega$ for a word $w \in \Sigma^*$), (ii) *deterministic BAs* (DBAs), (iii) *semideterministic BAs* (SDBAs; BAs 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* Σ. A (nondeterministic) *generalized Büchi automaton* (GBA) with k accepting sets is a tuple $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F})$, where Q is a finite set of states, $\delta\colon Q \times \Sigma \to 2^Q$ is a transition function, $Q_I \subseteq Q$ is a set of initial states, and $\mathcal{F} = \{\, F_j \subseteq Q \mid j \in \{1, \ldots, k\} \,\}$ 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 $q \xrightarrow{a} p$ to denote that $p \in \delta(q, a)$, and we define $post(q) = \bigcup_{a \in \Sigma} \delta(q, a)$. We lift δ to sets of states in the usual way. We abuse notation and for $q \in Q$ use $\mathcal{F}(q) = \{\, j \in \{1, \ldots, k\} \mid q \in F_j \,\}$ 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 $w = w_0 w_1 \ldots \in \Sigma^\omega$ 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 π 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 $i \geq 0$, $q_i \notin \bigcup_{1 \leq j \leq k} F_j$. A *run* $\rho = q_0 q_1 \ldots$ 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 $\mathcal{L}_\mathcal{A}(q) = \{w \in \Sigma^\omega \mid \mathcal{A}$ has an accepting trace on w 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 $\mathcal{L}(\mathcal{A}) = \bigcup_{q_i \in Q_I} \mathcal{L}(q_i)$. We use $\subseteq_\mathcal{L}$ to denote the relation of language inclusion of states: $p \subseteq_\mathcal{L} q \iff \mathcal{L}(p) \subseteq \mathcal{L}(q)$.

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, δ, 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 $\forall q \in Q, a \in \Sigma : |\delta(q, a)| \leq 1 \wedge |Q_I| = 1$. Moreover, $\mathcal{A}$ is a *semideterministic 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 $\delta = \delta_1 \cup \delta_t \cup \delta_2$ then consists of the following three disjoint transition functions: $\delta_1\colon Q_1 \times \Sigma \to 2^{Q_1}$, $\delta_t\colon Q_1 \times \Sigma \to 2^{Q_2}$, and $\delta_2\colon Q_2 \times \Sigma \to 2^{Q_2}$, where the relation δ_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., $\delta_t(Q_1, a) \subseteq F$ for all $a \in \Sigma$ and (2) we require that $Q_I \cap Q_2 \subseteq F$, 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 $(\bigcup_{a \in \Sigma} \delta_t(Q_1, a)) \setminus F$ or $(Q_I \cap Q_2) \setminus F$) 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 $\mathcal{A}$. The termination proof of P can be obtained by decomposing $\mathcal{A}$ into several BAs $\mathcal{A}_1, \ldots, \mathcal{A}_n$, whose languages jointly cover $\mathcal{L}(\mathcal{A})$, and then showing that each of them is terminating by means of a certified module [33].

Given a well-ordered set $(W, \prec)$, let ∞ 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* Φ 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 Σ 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 `i>0` contains the pairs (Φ, Φ) where $\Phi(\texttt{i}) \implies \texttt{i} > 0$. A *Hoare triple* is a triple $\{\psi\}\ \mathit{stmt}\ \{\psi'\}$ where stmt is a statement and ψ, ψ' are predicates over program variables; a Hoare triple is valid if, for each pair of valuations (Φ, Φ') in the relation associated with stmt, if Φ satisfies ψ, then Φ' satisfies ψ'.

Definition 3.1 (cf. [33, Definition 3])**.** Given a BA $\mathcal{A}_\mathcal{M} = (Q, \delta, \{q_i\}, \{q_f\})$ and a *ranking function* $f_\mathcal{M}$ from valuations into a well-ordered set $(W, \prec)$, we call a mapping $\mathcal{I}_\mathcal{M}$ from states to predicates over program variables a *rank certificate* for $f_\mathcal{M}$ and $\mathcal{A}_\mathcal{M}$ if the following properties hold:

- The initial state q_i is mapped by $\mathcal{I}_\mathcal{M}$ to the predicate where the auxiliary variable `oldrnk` has the value ∞, i.e., $\mathcal{I}_\mathcal{M}(q_i) \iff \texttt{oldrnk} = \infty$.
- The accepting state q_f is mapped by $\mathcal{I}_\mathcal{M}$ 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., $\mathcal{I}_\mathcal{M}(q_f) \implies f_\mathcal{M}(\vec{v}) \prec \texttt{oldrnk}$.
- Each outgoing transition $q \xrightarrow{stmt} q'$ from a state $q \neq q_f$ corresponds to a valid Hoare triple, i.e., the triple $\{\mathcal{I}_\mathcal{M}(q)\}\ \mathit{stmt}\ \{\mathcal{I}_\mathcal{M}(q')\}$ 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., $\{\mathcal{I}_\mathcal{M}(q_f)\}$ $\texttt{oldrnk:=}\ f_\mathcal{M}(\vec{v})$; stmt $\{\mathcal{I}_\mathcal{M}(q')\}$ is valid.

We call $\mathcal{M} = (\mathcal{A}_\mathcal{M}, f_\mathcal{M}, \mathcal{I}_\mathcal{M})$ a *certified module* and define its language as $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A}_\mathcal{M})$. A certified module represents a set of paths in P that share the same termination argument. That is, for all paths represented by $\mathcal{M}$, the evaluation of the ranking function $f_\mathcal{M}$ strictly decreases on visiting the accepting state q_f.

3.1 The Multi-Stage Approach to Construct $\mathcal{M}$

In this section, we describe our algorithm that generalizes an ultimately periodic word uv^ω 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* $\mathcal{M}_{uv^\omega}$ (cf. Section 3.1.1), which closely resembles the structure of uv^ω. The alphabet Σ of $\mathcal{M}_{uv^\omega}$ (and of its generalizations, see below) consists of all statements occurring in uv^ω. 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^ω.

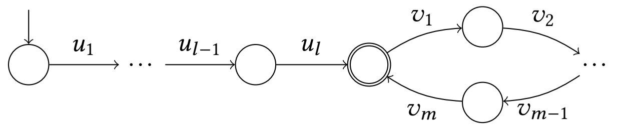
The previous work [33] uses a generalization procedure that uses $\mathcal{M}_{uv^\omega}$ to construct a module $\mathcal{M}_{nondet}$ consisting of a nondeterministic BA (cf. Section 3.1.5). Although $\mathcal{M}_{nondet}$ is usually quite general, the drawback of this solution is the extremely high 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 $\mathcal{M}_{uv^\omega}$ 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^ω (so that when its language is removed from the set of uncertified traces, we are guaranteed to remove at least uv^ω). 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^ω. As the last option, we construct $\mathcal{M}_{nondet}$, which is guaranteed to contain uv^ω in its language, but is the hardest to complement.

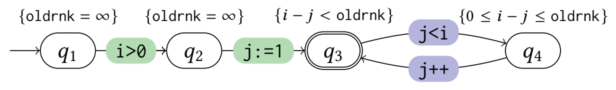
In this work we consider 4 stages: besides stage 0, where the initial certified lasso module $\mathcal{M}_{uv^\omega}$ is built, we have at stage 1 the *finite-trace certified module* construction (cf. Section 3.1.2). The result contains a finite-trace BA whose complementation takes constant time; the 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* $\mathcal{M}_{nondet}$ (cf. Section 3.1.5), which is guaranteed to accept uv^ω 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

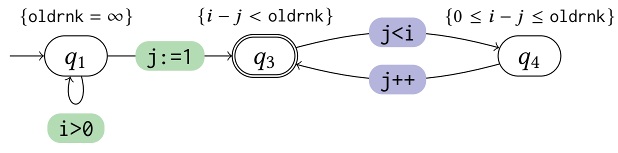
The *initial certified lasso module* $\mathcal{M}_{uv^\omega}$ consists of a BA accepting solely the word uv^ω, a ranking function f, and a rank certificate $\mathcal{I}$ (from a lasso program termination prover). The construction starts with the BA of the form depicted below, for the *stem* $u = u_1 u_2 \ldots u_l$ and the *loop* $v = v_1 v_2 \ldots v_m$.[1]



For instance, consider again the sorting program P^{sort} and the ω-word uv^ω = `i>0` `j:=1` (`j<i` `j++`)$^\omega$. The corresponding BA and rank certificate $\mathcal{I}^{\text{sort}}$ are depicted below, where each state is annotated with the corresponding predicate given by $\mathcal{I}^{\text{sort}}$ for the ranking function $f^{\text{sort}}(i,j) = i-j$.



Module $\mathcal{M}_{uv^\omega}$ is obtained by generalizing the constructed BA by merging states with the same predicate. Note that for the given word uv^ω, 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 `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^ω = `i>0` `j:=1` (`j<i` `j++`)$^\omega$.



We denote the module from the example above as $\mathcal{M}^{\text{sort}}_{uv^\omega}$.

3.1.2 Stage 1: Finite-trace Certified Module

The first module we try to construct is the *finite-trace certified module* $\mathcal{M}_1 = \mathcal{M}_{\text{fin}}$. This module can be constructed when uv^ω 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. $\mathcal{I}(q)$ is unsatisfiable and for every q' on the shortest path from q_i to q, $\mathcal{I}(q')$ is satisfiable. More concretely, $\mathcal{M}_{\text{fin}}$ can be constructed from $\mathcal{M}_{uv^\omega}$ 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 $q \xrightarrow{stmt} q$ for all $stmt \in \Sigma$, and (4) setting q as the single new accepting state.

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

3.1.3 Stage 2: Deterministic Certified Module

If $\mathcal{M}_1$ cannot be constructed, we proceed by building a *deterministic certified module* $\mathcal{M}_2 = \mathcal{M}_{\text{det}}$. The high-level intuitive idea is to construct a DBA using sets of states of $\mathcal{M}_{uv^\omega}$ with transitions that respect the predicates of $\mathcal{M}_{uv^\omega}$ (so that the termination argument for uv^ω correctly extends to the whole language of the module) and are in some sense *maximal*. In particular, the successor of a set of states $\mathfrak{Q}$ of $\mathcal{M}_{uv^\omega}$ over a statement $stmt$ is computed as the maximal set of states $\mathfrak{Q}'$ satisfying the following property: the predicate of every state in $\mathfrak{Q}'$ is a logical consequence of the conjunction of the predicates of the states in $\mathfrak{Q}$ and the semantics of $stmt$.

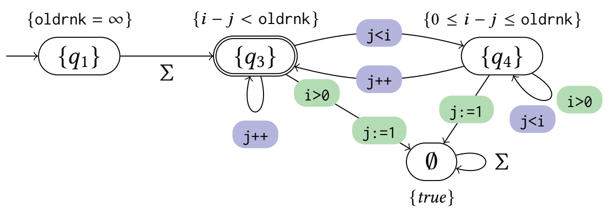
For instance, let us consider an initial certified lasso module $\mathcal{M}_{uv^\omega}$ whose alphabet contains the statement `z:=x+y` and whose states q_{23}, q_{42}, q_{65} are annotated by $\mathcal{I}$ as follows: $\mathcal{I}(q_{23})$ is $x = 23$, $\mathcal{I}(q_{42})$ is $y = 42$, and $\mathcal{I}(q_{65})$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

Definition 3.2. Let $\mathcal{M}_{uv^\omega} = (\mathcal{A}, f, \mathcal{I})$ be an initial certified lasso module such that $\mathcal{A} = (Q, \delta, \{q_i\}, \{q_f\})$. The *deterministic certified module* $\mathcal{M}_{\text{det}} = (\mathcal{A}^{\text{det}}, f, \mathcal{I}^{\text{det}})$ with a DBA $\mathcal{A}^{\text{det}} = (Q^{\text{det}}, \delta^{\text{det}}, {Q_I}^{\text{det}}, F^{\text{det}})$ is defined as follows:

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

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

By applying the certified deterministic module construction to $\mathcal{M}^{\text{sort}}_{uv^\omega}$, we obtain the following module:



Note that this module $\mathcal{M}^{\text{sort}}_{\text{det}}$, despite accepting a non-empty language, is absolutely useless for the refinement of P^{sort}, since it rejects the word uv^ω = `i>0` `j:=1` (`j<i` `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]).

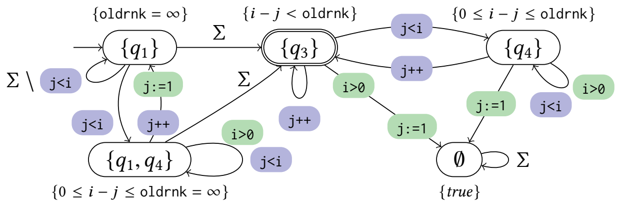
3.1.4 Stage 3: Semideterministic Certified Module

In order to overcome the shortcomings of the deterministic certified module, we now present the *semideterministic certified module* $\mathcal{M}_3 = \mathcal{M}_{\text{semi}}$, which is $\mathcal{M}_{\text{det}}$ enriched with additional transitions. In particular, for a statement *stmt*, each state $\mathfrak{Q}$ that is not reachable from an accepting state s.t. $q_f \in \delta_\wedge(\mathfrak{Q}, stmt)$ (cf. Definition 3.2) has two *stmt*-successors:

- $\delta_\wedge(\mathfrak{Q}, stmt) \setminus \{ q \in Q \mid q \neq q_f \wedge \texttt{oldrnk} \in \text{var}(\mathcal{I}(q)) \}$, the original successor in δ^{det};
- an additional successor $\delta_\wedge(\mathfrak{Q}, stmt) \setminus \{q_f\}$.

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 $\texttt{oldrnk} = \infty$, while all states in the loop imply $\texttt{oldrnk} < \infty$).

By applying the certified semideterministic module construction to $\mathcal{M}^{\text{sort}}_{uv^\omega}$, we obtain the following module $\mathcal{M}^{\text{sort}}_{\text{semi}}$.



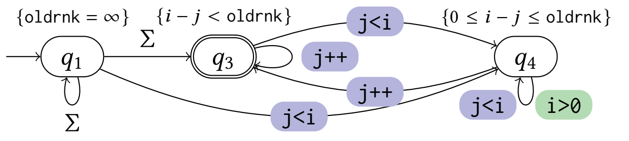
Note that, in contrast to $\mathcal{M}^{\text{sort}}_{\text{det}}$, the module $\mathcal{M}^{\text{sort}}_{\text{semi}}$ already accepts the word uv^ω = `i>0` `j:=1` (`j<i` `j++` $)^\omega$.

Note that although the construction of the semideterministic module can theoretically produce an exponential-sized automaton, we rarely experienced this case in our evaluation.

3.1.5 Stage 4: Nondeterministic Certified Module

The *nondeterministic certified module* $\mathcal{M}_4 = \mathcal{M}_{nondet}$ is the most powerful generalization we considered. It is obtained by adding every possible transition between each pair of states to $\mathcal{M}_{uv^\omega}$, as long as the rank certificate is still correct.

For instance, the above lasso module $\mathcal{M}^{\text{sort}}_{uv^\omega}$ becomes the following nondeterministic module.



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

Although $\mathcal{M}_{nondet}$ is always guaranteed to accept uv^ω (since it contains all transitions of $\mathcal{M}_{uv^\omega}$), its use in the overall termination procedure is expensive, because algorithms for complementing BAs have a prohibitive complexity. Based on our experiments, constructing $\mathcal{M}_{nondet}$ is seldom necessary, and in many cases, $\mathcal{M}_{\text{semi}}$ 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 $\mathcal{A}$ (in our setting representing program paths whose termination has not yet been established) and, in general, a BA $\mathcal{B}$ (which represents the program paths whose termination we have just proved), constructs a GBA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$. 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 $\mathcal{B}$, further denoted as $\overline{\mathcal{B}}$, then constructs a GBA $\mathcal{A}_I$ accepting the intersection $\mathcal{L}(\mathcal{A}_I) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}})$ and, finally, removes useless states from $\mathcal{A}_I$ (yielding an empty automaton in the case $\mathcal{L}(\mathcal{A}_I) = \emptyset$). Recall that a state q is useless iff $\mathcal{L}_{\mathcal{A}_I}(q) = \emptyset$, otherwise, q is *useful*. Our optimizations that make the algorithm usable in practice are the following.

1. $\overline{\mathcal{B}}$ is constructed *on the fly* when constructing $\mathcal{A}_I$, i.e., only those states of $\overline{\mathcal{B}}$ that occur in some product state of $\mathcal{A}_I = \mathcal{A} \cap \overline{\mathcal{B}}$ 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

```
Input   :GBA $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F})$
Output :GBA $\mathcal{A}' = (Q', \delta', Q_I', \mathcal{F}')$ s.t. $\forall q \in Q' : \mathcal{L}(q) \neq \emptyset$
Global :$Q' \leftarrow \emptyset$, emp $\leftarrow \emptyset$, SCCs $\leftarrow \emptyset$, act $\leftarrow \emptyset$, cnt $\leftarrow 0$
1  Function remove_useless($\mathcal{A}$):
2      foreach $q_I \in I$ do
3          if $q_I \notin Q' \cup$ emp then    // $q_I \notin Q' \cup \lceil$emp$\rceil$
4              construct($q_I$);
5      return $\mathcal{A}' = (Q', \delta \cap (Q' \times \Sigma \times 2^{Q'}), I \cap Q', \mathcal{F}_{|Q'})$;

6  Function construct($s$):
7      ++cnt; $s$.dfsnum $\leftarrow$ cnt; is_nemp $\leftarrow$ false;
8      SCCs.push(($s, \mathcal{F}(s)$)); act.push($s$);
9      foreach $t \in post(s)$ do
10         if $t \in Q'$ then is_nemp $\leftarrow$ true;
11         else if $t \in$ emp then continue; // $t \in \lceil$emp$\rceil$
12         else if $t \notin$ act then
13             is_nemp $\leftarrow$ construct($t$) $\vee$ is_nemp;
14         else
15             B $\leftarrow \emptyset$;
16             do
17                 $(u, C) \leftarrow$ SCCs.pop(); $B \leftarrow B \cup C$;
18                 if $B = \mathcal{F}$ then is_nemp $\leftarrow$ true ;
19             while $u$.dfsnum $> t$.dfsnum;
20             SCCs.push(($u, B$));
21     if SCCs.top() = ($s$, _) then
22         SCCs.pop();
23         do
24             $u \leftarrow$ act.pop();
25             if is_nemp then $Q'$.add($u$);
26             else emp.add($u$);
27         while $u \neq s$;
28     return is_nemp;
```

2. We remove useless states from $\mathcal{A}_I$ 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 $\mathcal{B}$ is an SDBA, we optimize the construction of $\overline{\mathcal{B}}$ 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 $\mathcal{A}_I$ (Section 6).

4.1 Removing Useless States in a GBA

Algorithm 1 is a modification of the algorithm for checking emptiness of a GBA $\mathcal{A} = (Q, \delta, Q_I, \mathcal{F} = \{F_1, \ldots, F_k\})$ 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 F_j. Our modification not only tests the emptiness of $\mathcal{L}(\mathcal{A})$, but also efficiently constructs a copy $\mathcal{A}'$ of $\mathcal{A}$ without any useless state (and, therefore, if $\mathcal{L}(\mathcal{A}) = \emptyset$, then $\mathcal{A}'$ 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 constituting 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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_e the set of accepting conditions from $\mathcal{F}$ that q_e can infinitely many times reach ($\mathcal{F}(s) \subseteq \{1, \ldots, k\}$ denotes all accepting conditions that s belongs to).

The differences of Algorithm 1 from GS are the following: (i) Algorithm 1 does not stop immediately when an accepting SCC is found (line 18), but continues in the construction, (ii) 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 (iii) 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 $\mathcal{A}$ 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 $\mathcal{A}$, i.e., $\mathcal{A}$ 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 $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\overline{\mathcal{B}})$, which requires an efficient construction of $\overline{\mathcal{B}}$. In this section, we present such a construction for an SDBA $\mathcal{B}$.

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 $\mathcal{A}_C = (Q_C, \delta_C, Q_{I_C}, F_C)$ a *macro-run*. We usually denote states in Q_C as $\widehat{q} = (N, C, S, B)$, where the components have the following intuitive meaning:

- N (nondeterministic): If a run of $\mathcal{A}$ touches a state in N, then it is still in the nondeterministic part Q_1.
- C (choice): If a run of $\mathcal{A}$ 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 $\mathcal{A}_C$ as soon as it attempts to touch an accepting state, i.e., if $q \in S$ and $\delta_2(q, a) \in F$, then $\delta_C(\widehat{q}, a) = \emptyset$ (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 $\mathcal{A}$ 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 $\mathcal{A} = (Q_1 \cup Q_2, \delta, Q_I, F)$, where Q_1 and Q_2 are defined as in Section 2, its complement automaton $\mathcal{A}_C = (Q_C, \delta_C, Q_{I_C}, F_C)$ is defined as follows:

- $Q_C = \{(N, C, S, B) \in 2^{Q_1} \times 2^{Q_2} \times 2^{Q_2 \setminus F} \times 2^{Q_2} \mid B \subseteq C\}$.
- $Q_{I_C} = \{(Q_1 \cap Q_I, Q_2 \cap Q_I, \emptyset, Q_2 \cap Q_I)\}$.
- $F_C = \{\, (N, C, S, B) \in Q_C \mid B = \emptyset \,\}$.
- δ_C is the transition function $\delta_C : Q_C \times \Sigma \to 2^{Q_C}$ such that $(N', C', S', B') \in \delta_C((N, C, S, B), a)$ iff
  1. $N' = \delta_1(N, a)$,
  2. $C' \cup S' = \delta_t(N, a) \cup \delta_2(C \cup S, a)$,
  3. $C' \cap S' = \emptyset$,
  4. $S' \supseteq \delta_2(S, a)$,
  5. $C' \supseteq \delta_2(C \setminus F, a)$, and
  6. $B' = C'$ if $B = \emptyset$, otherwise $B' = \delta_2(B, a) \cap C'$.

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 $\delta_t(N, a) \cup \delta_2(C \cup S, a)$ stay in C', because S' is a set of non-accepting states, and (4) the rest of the states in $\delta_t(N, a) \cup \delta_2(C \cup S, a)$ 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 $\delta_2(q, a) \neq \emptyset$." 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 clearer). When talking about the size of the set of states or transitions, we only consider those states and transitions reachable from Q_{I_C}.

The best way to get an intuition about the algorithm is to simulate both accepting and rejecting runs of $\mathcal{A}$ in $\mathcal{A}_C$. Let $\rho = q_0 q_1 \ldots q_i \ldots$ be an accepting run of $\mathcal{A}$ over some word $w \in \Sigma^\omega$ and q_i be the first accepting state in ρ. 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 ρ is moved from N to C at position i (rule 2 and the fact that S is disjoint with F), i.e., $q_k \in N_k$ 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 ρ is moved from C to S at a position $j + 1$. In this case, Π will be blocked later at the position of the next occurrence of an accepting state in ρ (which there are infinitely many), because once ρ has moved to S, it will stay in S (rule 4). It follows that Π is finite and, therefore, not an accepting macro-run.
- Case (2): If we assume that ρ stays in C for all such positions j, then (rule 6) the run ρ will be copied to B the next time B becomes empty (if it ever happens). But then B cannot become empty again because ρ will stay inside it forever (our assumption is that ρ stays in C forever and hence also in B by rule 6). It follows that although Π is infinite, it is not an accepting macro-run.

On the other hand, we can show that if $w \notin \mathcal{L}(\mathcal{A})$, we can construct from its rejecting runs $\rho = q_0 q_1 \ldots q_i \ldots q_j \ldots$ 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 $\mathcal{A}_C$. The strategy of the construction is simple. All such ρ 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 δ_2, there is only one corresponding run for each of them.

More concretely, we again have two cases: (i) if the run ρ never touches an accepting state, then ρ 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 ρ (it can happen that $i = j$), then there is a macro-run where ρ 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 Π:

1. Π is non-blocking. From the definition of δ_C, a macro-state can become blocking only in one of two cases: (1) 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 Π 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 Π is non-blocking.

2. Π contains infinitely many accepting macro-states. Starting from any macro-state of Π, no new runs can be moved to B until it becomes empty (rule 6). Since all runs ρ 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 $\boxed{\emptyset,\{q'_1,q'_2\},\{q'''_3\},\emptyset}$. In fact, the construction needs to know whether ρ 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 □) that are redundant because the guessing of whether to keep a run ρ 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 left-most 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 $\boxed{\emptyset,\{q'_1,q'_2\},\{q'''_3\},\emptyset}$. Having arrived at this macro-state, the guessing of all states in C have 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 δ_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. $C' \cup S' = \delta_t(N,a) \cup \delta_2(C \cup S,a)$,
3. $C' \cap S' = \emptyset$,
4. **[new]** $S' \supseteq \delta_2(S,a)$ if $B = \emptyset$, else $S' = \delta_2(S,a)$, and
5. **[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 ρ 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 □ 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 ρ in B has no chance to be moved to S, even for the case when ρ has no accepting states after the state in B. In such a case, ρ 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.

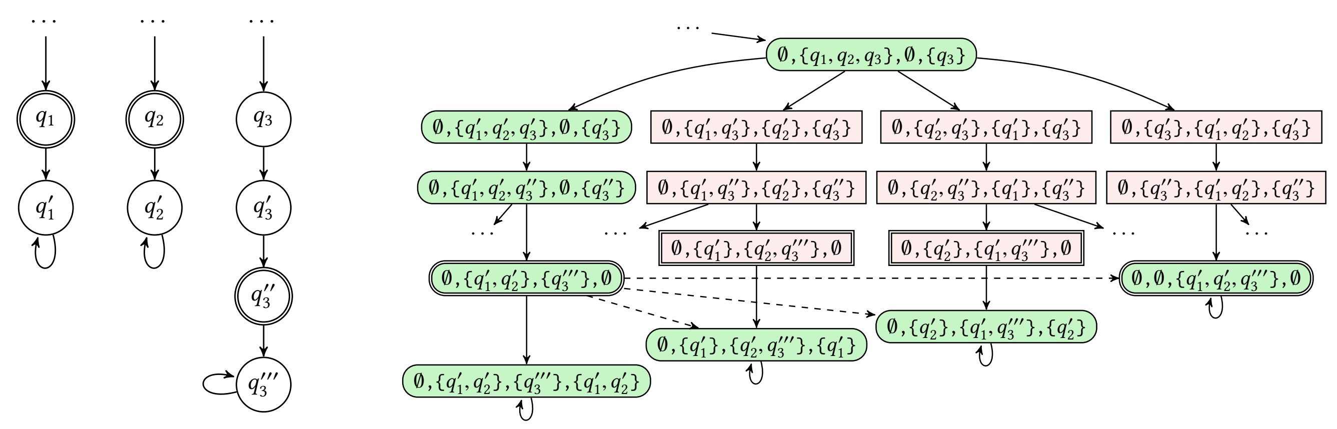
5.3 The NCSB-Lazy Algorithm

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 δ_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. $C' \cup S' = \delta_t(N,a) \cup \delta_2(C \cup S,a)$,
  a3. $C' \cap S' = \emptyset$,
  a4. $S' \supseteq \delta_2(S,a)$, and
  a5. **[removed]**
  a6. $B' = C'$.
- When $B \neq \emptyset$:
  b1. $N' = \delta_1(N,a)$,
  b2. **[new]** $B' \cup S' = \delta_2(B \cup S,a)$,
  b3. **[new]** $B' \cap S' = \emptyset$,
  b4. $S' \supseteq \delta_2(S,a)$,
  b5. **[new]** $C' = (\delta_2(C,a) \cup \delta_t(N,a)) \setminus S'$, and
  b6. **[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.*


q_1
q_2
q_3
q_1'
q_2'
q_3'
q_3''
q_3'''
$\emptyset, \{q_1, q_2, q_3\}, \emptyset, \{q_3\}$
$\emptyset, \{q_1', q_2', q_3'\}, \emptyset, \{q_3'\}$
$\emptyset, \{q_1', q_3'\}, \{q_2'\}, \{q_3'\}$
$\emptyset, \{q_2', q_3'\}, \{q_1'\}, \{q_3'\}$
$\emptyset, \{q_3'\}, \{q_1', q_2'\}, \{q_3'\}$
$\emptyset, \{q_1', q_2', q_3''\}, \emptyset, \{q_3''\}$
$\emptyset, \{q_1', q_3''\}, \{q_2'\}, \{q_3''\}$
$\emptyset, \{q_2', q_3''\}, \{q_1'\}, \{q_3''\}$
$\emptyset, \{q_3''\}, \{q_1', q_2'\}, \{q_3''\}$
$\emptyset, \{q_1'\}, \{q_2', q_3'''\}, \emptyset$
$\emptyset, \{q_2'\}, \{q_1', q_3'''\}, \emptyset$
$\emptyset, \{q_1', q_2'\}, \{q_3'''\}, \emptyset$
$\emptyset, \emptyset, \{q_1', q_2', q_3'''\}, \emptyset$
$\emptyset, \{q_2'\}, \{q_1', q_3'''\}, \{q_2'\}$
$\emptyset, \{q_1'\}, \{q_2', q_3'''\}, \{q_1'\}$
$\emptyset, \{q_1', q_2'\}, \{q_3'''\}, \{q_1', q_2'\}$

(a) A part of $\mathcal{A}$ (we assume that all shown states are in Q_2).

(b) A part of the complemented automaton $\mathcal{A}_C$. Here we only draw macro-states (N, C, S, B) starting from $(\emptyset, \{q_1, q_2, q_3\}, \emptyset, \{q_3\})$ and omit the macro-states that keep q_3''' in C.

Figure 3. An example of inefficiency of eager guessing in NCSB-Original (we assume all transitions are over the symbol $a \in \Sigma$)

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 $\widehat{p} = (N, C, S, B)$ from $\mathcal{A}_C$. Further, assume that for all runs ρ over w in $\mathcal{A}$, it holds that*

1. *if ρ starts from a state $q \in N \cup C \cup S$, it is rejecting,*
2. *if ρ starts from a state $q \in S$, it is safe, and*
3. *if ρ 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 Π from $\widehat{p}$ 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 Π are deterministic, i.e., one can construct deterministically every macro-state in Π following the transition relation of NCSB-Lazy. We now show that Π is an accepting macro-run in $\mathcal{A}_C$ by proving two properties.

1. Π 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, Π 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. Π contains infinitely many accepting macro-states. Starting from any macro-state of Π, no new runs can be moved to B until it becomes empty (rule a6). Since all runs ρ of $\mathcal{A}$ on w are rejecting starting from any states in $\widehat{p}$, 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 $\mathcal{L}(\mathcal{A}_C) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

Proof. The case that $w \in \mathcal{L}(\mathcal{A})$ implies $w \notin \mathcal{L}(\mathcal{A}_C)$ can be proved in a similar way as in NCSB-Original. In particular, we need to show that any accepting run ρ of $\mathcal{A}$ over w will either stay forever in B or move to S and block the macro-run.

For the case that $w \notin \mathcal{L}(\mathcal{A})$ implies $w \in \mathcal{L}(\mathcal{A}_C)$, we can construct an accepting macro-run Π from the runs ρ of $\mathcal{A}$ on w using Lemma 5.3. In order to do so, we need to ensure that the initial macro-state $(Q_1 \cap Q_I, Q_2 \cap Q_I, \emptyset, Q_2 \cap Q_I)$ satisfies the requirements of Lemma 5.3, i.e., for all runs ρ over w in $\mathcal{A}$, it holds that

1. if ρ starts from a state $q \in N \cup C \cup S$, it is rejecting,
2. if ρ starts from a state $q \in S$, it is safe, and
3. if ρ 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_I. Requirement 2 is satisfied because $S = \emptyset$. Requirement 3 is satisfied because all states in $Q_2 \cap Q_I$ 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* [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 $\widehat{p} = (N_p, C_p, S_p, B_p)$ and $\widehat{r} = (N_r, C_r, S_r, B_r)$, we define the following two subsumption relations:

$$\widehat{p} \sqsubseteq \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad N_p \supseteq N_r \land C_p \supseteq C_r \land S_p \supseteq S_r \text{ and} \tag{4}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \widehat{p} \sqsubseteq \widehat{r} \land B_p \supseteq B_r. \tag{5}$$

Let $\mathcal{A}^O_C$ and $\mathcal{A}^L_C$ be the complement automata constructed using NCSB-Original and NCSB-Lazy, respectively. We define two language inclusion relations $\subseteq^O_{\mathcal{L}}$ and $\subseteq^L_{\mathcal{L}}$ over macro-states $\widehat{p}$ and $\widehat{r}$ as follows:

$$\widehat{p} \subseteq^O_{\mathcal{L}} \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathcal{L}_{\mathcal{A}^O_C}(\widehat{p}) \subseteq \mathcal{L}_{\mathcal{A}^O_C}(\widehat{r}), \tag{6}$$

$$\widehat{p} \subseteq^L_{\mathcal{L}} \widehat{r} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathcal{L}_{\mathcal{A}^L_C}(\widehat{p}) \subseteq \mathcal{L}_{\mathcal{A}^L_C}(\widehat{r}). \tag{7}$$

The main result of this section is that for any macro-states $\widehat{p}$ and $\widehat{r}$, the following implications hold:

$$\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p} \subseteq^O_{\mathcal{L}} \widehat{r} \qquad \text{(Section 6.1) and} \tag{8}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \implies \widehat{p} \subseteq^L_{\mathcal{L}} \widehat{r} \qquad \text{(Section 6.2).} \tag{9}$$

As a consequence, we can use $\sqsubseteq$ and $\sqsubseteq^B$ in Algorithm 1 (when computing the difference automaton $\mathcal{A} \setminus \mathcal{B}$) 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 $\widehat{q}$ in the set emp into testing the same in the set $\lceil \text{emp} \rceil$ defined as

$$\lceil \text{emp} \rceil = \{(q_{\mathcal{A}}, \widehat{q_{\mathcal{B}}}) \mid \exists (q_{\mathcal{A}}, \widehat{r_{\mathcal{B}}}) \in \text{emp} : \widehat{q_{\mathcal{B}}} \sqsubseteq' \widehat{r_{\mathcal{B}}}\}, \tag{10}$$

where $\sqsubseteq' \in \{\sqsubseteq, \sqsubseteq^B\}$ depending on the particular algorithm used for complementation ($\sqsubseteq$ for NCSB-Original and $\sqsubseteq^B$ for NCSB-Lazy). Note that on line 26, emp can be maintained in the form of an antichain, i.e., to contain only elements incomparable w.r.t. $\sqsubseteq'$.

6.1 Subsumption Relation for NCSB-Original

We first show that $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p} \subseteq^O_{\mathcal{L}} \widehat{r}$ for any two macro-states $\widehat{p} = (N_p, C_p, S_p, B_p)$ and $\widehat{r} = (N_r, C_r, S_r, B_r)$. We prove this fact by constructing a *strategy* that for any accepting macro-run from $\widehat{p}$ returns an accepting macro-run from $\widehat{r}$ 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 $\sqsubseteq$ and $\sqsubseteq^B$ are instances of the corresponding early simulation relations.

6.1.1 Early Simulation

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

$$\forall i < j : ((p_i \in F \lor i = -1) \land p_j \in F) \implies \exists i < k \leq j : r_k \in F, \tag{11}$$

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

$$\forall i < j : (p_i, p_j \in F) \implies \exists i < k \leq j : r_k \in F. \tag{12}$$

Intuitively, the *early+1* simulation requires that between every two times π_p touches an accepting state, π_r also touches an accepting state; the *early* simulation further requires that π_r first touches an accepting state not later than π_p does.

We extend the proposed notions of simulation to states as follows. First, we define a *strategy* as a function $\sigma : Q \times (Q \times \Sigma \times Q) \to (Q \times \Sigma \times Q)$ such that $\sigma(r, p \xrightarrow{a} p') = r \xrightarrow{a} r'$ where $r' \in \delta(r, a)$. That is, σ picks a transition from r based on the transition $p \xrightarrow{a} p'$ selected by the environment. Next, we lift strategy to traces such that for a trace π_p defined as above, we set $\sigma(r_0, \pi_p) = r_0 \xrightarrow{w_0} r_1 \xrightarrow{w_1} \cdots$ where for all $i \geq 0$ it holds that $\sigma(r_i, p_i \xrightarrow{w_i} p_{i+1}) = r_i \xrightarrow{w_i} r_{i+1}$. We say that p_0 is early (resp. early+1) simulated by r_0, denoted as $p_0 \preceq_e r_0$ (resp. $p_0 \preceq_{e^{+1}} r_0$) iff there exists a strategy σ_e (resp. $\sigma_{e^{+1}}$) such that for every trace π_p starting in p_0, it holds that $\pi_p \preceq_e \sigma_e(r_0, \pi_p)$ (resp. $\pi_p \preceq_{e^{+1}} \sigma_{e^{+1}}(r_0, \pi_p)$).

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

Proposition 6.1. *Given a BA $\mathcal{A}$, the following holds for the relations over the states of $\mathcal{A}$:*

$$\preceq_e \quad \subseteq \quad \preceq_{e^{+1}} \quad \subseteq \quad \subseteq_{\mathcal{L}}. \tag{13}$$

6.1.2 The Subsumption $\sqsubseteq$ is an Early Simulation

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

Lemma 6.2. *The relations $\sqsubseteq$ and $\sqsubseteq^B$ on $\mathcal{A}^O_C$ are an early+1 simulation and an early simulation respectively:*

$$\widehat{p} \sqsubseteq \widehat{r} \quad \implies \quad \widehat{p} \preceq_{e^{+1}} \widehat{r} \quad \textit{and} \tag{14}$$

$$\widehat{p} \sqsubseteq^B \widehat{r} \quad \implies \quad \widehat{p} \preceq_e \widehat{r}. \tag{15}$$

Proof of (14). We use the strategy $\sigma_{\sqsubseteq}$ that for a transition $\widehat{p} \xrightarrow{a} \widehat{p'} = (N_{p'}, C_{p'}, S_{p'}, B_{p'})$ chooses a transition $\widehat{r} \xrightarrow{a} \widehat{r'} = (N_{r'}, C_{r'}, S_{r'}, B_{r'})$ that respects all nondeterministic choices made in $\widehat{p} \xrightarrow{a} \widehat{p'}$. In particular, if the successor q' of a state $q \in C_p \cap C_r$ was moved to $S_{p'}$, i.e., $q' = \delta_2(q, a) \in S_{p'}$, the strategy $\sigma_{\sqsubseteq}$ will also move q' to $S_{r'}$, otherwise q' will stay in $C_{r'}$. Other parts of the construction of $\widehat{r'}$ are deterministic (i.e., just follow the definition of δ_C in Definition 5.1). The strategy guarantees that (1) $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p}' \sqsubseteq \widehat{r}'$ and (2) the transition $\widehat{r} \xrightarrow{a} \widehat{r}'$ exists (proof omitted due to lack of space).

Next, we show that for any two traces $\pi_p = \widehat{p_0} \xrightarrow{w_0} \widehat{p_1} \xrightarrow{w_1} \cdots$ and $\pi_r = \sigma_{\sqsubseteq}(r_0, \pi_p) = \widehat{r_0} \xrightarrow{w_0} \widehat{r_1} \xrightarrow{w_1} \cdots$, such that $\widehat{p_0} = \widehat{p}$ and $\widehat{r_0} = \widehat{r}$, the condition $\pi_p \preceq_{e^{+1}} \pi_r$ is satisfied, i.e., $\forall i < j : (\widehat{p_i}, \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O$ holds, where F^O is the set of accepting macro-states of $\mathcal{A}^O_C$.

Claim: For all $i \geq 0$, if $\widehat{p_i} \in F^O$, then $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$.

Proof: Let the macro-states $\widehat{p_{i+1}}$ and $\widehat{r_{i+1}}$ be as follows: $\widehat{p_{i+1}} = (N_{p_{i+1}}, C_{p_{i+1}}, S_{p_{i+1}}, B_{p_{i+1}})$ and $\widehat{r_{i+1}} = (N_{r_{i+1}}, C_{r_{i+1}}, S_{r_{i+1}}, B_{r_{i+1}})$. The following holds: (i) $B_{p_{i+1}} = C_{p_{i+1}}$ (because $\widehat{p_i} \in F^O$), (ii) $C_{p_{i+1}} \supseteq C_{r_{i+1}}$ (due to the property of $\sigma_{\sqsubseteq}$, i.e., $\widehat{p} \sqsubseteq \widehat{r} \implies \widehat{p}' \sqsubseteq \widehat{r}'$), and (iii) $C_{r_{i+1}} \supseteq B_{r_{i+1}}$ (the property of a macro-state). It follows that $B_{p_{i+1}} \supseteq B_{r_{i+1}}$ and hence $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$. ■

Claim: If $\widehat{p_{i+1}} \sqsubseteq^B \widehat{r_{i+1}}$ and $\widehat{p_j} \in F^O$ for some $i < j$, then there exists some k such that $i < k \leq j$ and $\widehat{r_k} \in F^O$.

Proof: Since $B_{p_{i+1}} \supseteq B_{r_{i+1}}$ and due to the property of $\sigma_{\sqsubseteq}$ that every state that is moved from $B_{p_{i+1}}$ to $S_{p_{i+1}}$ in π_p will be by $\sigma_{\sqsubseteq}$ also simultaneously moved from $B_{r_{i+1}}$ to $S_{r_{i+1}}$ in π_r, the set $B_{r_{i+1}}$ in π_r will become empty not later than $B_{p_{i+1}}$ becomes empty in π_p. ■

The two claims above imply that $\pi_p \preceq_{e^{+1}} \pi_r$. □

Proof of (15). We use the same strategy $\sigma_{\sqsubseteq}$ from the proof of (14). We show that for any two traces $\pi_p = \widehat{p_0} \xrightarrow{w_0} \widehat{p_1} \xrightarrow{w_1} \cdots$ and $\pi_r = \sigma_{\sqsubseteq}(r_0, \pi_p) = \widehat{r_0} \xrightarrow{w_0} \widehat{r_1} \xrightarrow{w_1} \cdots$, it follows that $\pi_p \preceq_e \pi_r$, i.e., $\forall i < j : ((\widehat{p_i} \in F^O \lor i = -1) \land \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O$, where F^O is the set of accepting macro-states of $\mathcal{A}^O_C$. First, we change $\pi_p \preceq_e \pi_r$ into an equivalent conjunction of the following two conditions:

$$\forall i < j : (\widehat{p_i}, \widehat{p_j} \in F^O) \implies \exists i < k \leq j : \widehat{r_k} \in F^O, \quad (16)$$

$$\widehat{p_i} \in F^O \implies \exists k \leq i : \widehat{r_k} \in F^O. \quad (17)$$

We notice that Condition (16) is equivalent to $\pi_p \preceq_{e^{+1}} \pi_r$ and since $\sqsubseteq^B$ is stronger than $\sqsubseteq$, from (14) it follows that $\widehat{p} \preceq_{e^{+1}} \widehat{r}$, and because the strategy $\sigma_{\sqsubseteq}$ in the proof of (14) is the same, Condition (16) also holds. Condition (17), on the other hand, follows from the second claim in the proof of (14). □

The following theorem states that $\sqsubseteq$ and $\sqsubseteq^B$ are subsumption relations over the macro-states of $\mathcal{A}^O_C$.

Theorem 6.3. *The relations $\sqsubseteq^B$ and $\sqsubseteq$ under-approximate language inclusion of macro-states in a complement automaton constructed using NCSB-Original:*

$$\sqsubseteq^B \quad \subseteq \quad \sqsubseteq \quad \subseteq \quad \subseteq^O_{\mathcal{L}}. \quad (18)$$

Proof. Follows from Proposition 6.1 and Lemma 6.2. □

6.2 Subsumption relation for NCSB-Lazy

The BAs produced by NCSB-Lazy are different from the BAs produced by NCSB-Original. This, in particular, means that the subsumption relation $\sqsubseteq$ does not under-approximate the language inclusion $\subseteq^L_{\mathcal{L}}$ in BAs produced by NCSB-Lazy.

Remark: Let $\mathcal{A}^L_C$ by a BA produced by NCSB-Lazy from $\mathcal{A}$ and $\widehat{p}$ and $\widehat{r}$ be a pair of macro-states of $\mathcal{A}^L_C$. In general, $\widehat{p} \sqsubseteq \widehat{r}$ does not imply $\widehat{p} \subseteq^L_{\mathcal{L}} \widehat{r}$. In particular, let q be a non-accepting state of $\mathcal{A}$ with only one outgoing transition that is a self-loop and let $\widehat{p} = (\emptyset, \{q\}, \emptyset, \emptyset)$ and $\widehat{r} = (\emptyset, \{q\}, \emptyset, \{q\})$. Note that $\widehat{p} \sqsubseteq \widehat{r}$ (as $\sqsubseteq$ does not relate the B components). Also note that there exists an accepting macro-run $\widehat{p} \cdot (\emptyset, \emptyset, \{q\}, \emptyset)^\omega$ from $\widehat{p}$, while there exists no accepting macro-run from $\widehat{r}$, since the B component of $\widehat{r}$ can never become empty (cf. condition b6 in Section 5.3). □

Fortunately, the stronger subsumption relation $\sqsubseteq^B$ still under-approximates the language inclusion $\subseteq^L_{\mathcal{L}}$, so it can be used to optimize the difference automaton construction.

Theorem 6.4. *The relation $\sqsubseteq^B$ under-approximates language inclusion of macro-states in a complement automaton constructed using NCSB-Lazy:*

$$\sqsubseteq^B \quad \subseteq \quad \subseteq^L_{\mathcal{L}}. \quad (19)$$

Proof. Let $\widehat{p} = (N, C, S, B)$ and $\widehat{r} = (N', C', S', B')$ s.t. $\widehat{p} \sqsubseteq^B \widehat{r}$. For any word $w \in \mathcal{L}_{\mathcal{A}^L_C}(\widehat{p})$, 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.

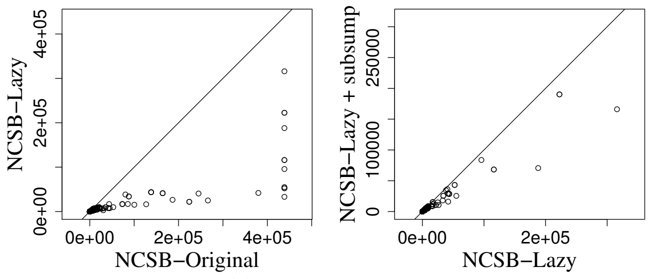
Hence, we can apply Lemma 5.3 to obtain an accepting macro-run from $\widehat{r}$ over w in $\mathcal{A}^L_C$. It follows that $\widehat{p} \subseteq^L_{\mathcal{L}} \widehat{r}$. □

7 Experimental Evaluation

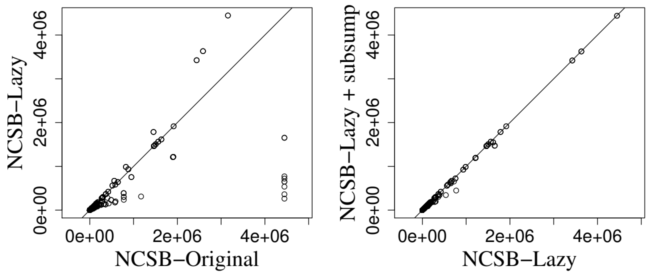
We implemented the presented techniques as an extension of Ultimate Automizer [33] and experimentally evaluated their performance. The results are presented in Figures 4 and 5. The points in the top/right-most regions of the plot represent experiments where the corresponding setting timed out (>300 s) 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 (1375 programs) in the Termination category of SV-Comp [1, 10][2].

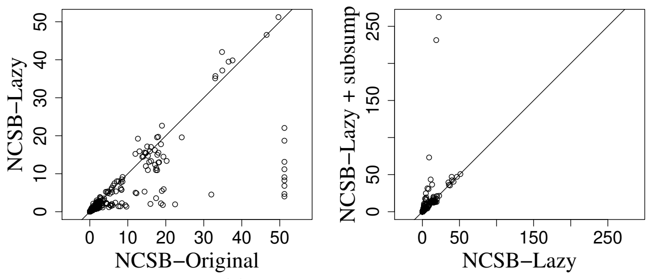
[2]We used 6 different settings and collected all SDBAs produced before the timeout. Because Ultimate Automizer constructs the difference of



(a) Number of states



(b) Number of transitions



(c) Execution time [s]

Figure 4. Comparing the performance of NCSB-Lazy with NCSB-Original and evaluating the effect of subsumption

Figure 4a shows that NCSB-Lazy significantly improves the number of states of the complemented automata, and that the subsumption can save even more states. In Figure 4b, we see that in the majority of cases, NCSB-Lazy also reduces the number of transitions. This is not guaranteed though; in several cases, the number of transitions increased. Subsumption is also not so helpful in reducing the number of transitions. In Figure 4c, we observe that in most cases, NCSB-Lazy also reduces the execution time. On the other hand, subsumption does not help that much as it brings significant overhead. Nevertheless, subsumption always produces fewer states in the BA language difference operation, which is an important factor for the overall performance of the termination analysis. More precisely, the average numbers of **S**tates and **T**ransitions for the three settings are the following:

NCSB-Original:	4,700 S	and	122,200 T
NCSB-Lazy:	2,900 S	and	132,300 T
NCSB-Lazy + Subsumption:	1,600 S	and	111,700 T

automata on the fly, if the construction does not finish before the timeout, the SDBA is not fully built and so cannot be used for this experiment.

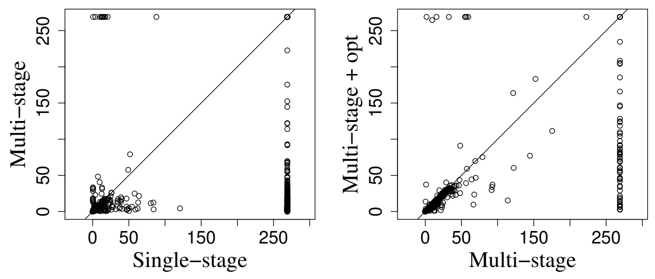


Figure 5. Evaluating the performance of the multi-stage approach and the optimized difference operation [s]

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 $\mathcal{M}_{nondet}$). For the multi-stage approach, we first evaluated three different generalization sequences:

(i) $\mathcal{M}_{uv^\omega} \rightarrow \mathcal{M}_{\text{fin}} \rightarrow \mathcal{M}_{\text{semi}} \rightarrow \mathcal{M}_{nondet}$ (we skip $\mathcal{M}_{\text{det}}$)
(ii) $\mathcal{M}_{uv^\omega} \rightarrow \mathcal{M}_{\text{fin}} \rightarrow \mathcal{M}_{\text{det}} \rightarrow \mathcal{M}_{nondet}$ (we skip $\mathcal{M}_{\text{semi}}$)
(iii) $\mathcal{M}_{uv^\omega} \rightarrow \mathcal{M}_{\text{fin}} \rightarrow \mathcal{M}_{\text{det}} \rightarrow \mathcal{M}_{\text{semi}} \rightarrow \mathcal{M}_{nondet}$

All of them solved roughly the same amount of examples (± 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 $\mathcal{M}_{nondet}$, 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 focussed 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 *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 *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, a ranking function of this paths 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/SAFEA 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

[1] Software Verification Competition (SV-Comp) Benchmarks. https://github.com/sosy-lab/sv-benchmarks. Accessed: 2017-11-01.

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. 2010. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In *Proceedings of 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. Springer-Verlag, Berlin, Heidelberg, 158–174. https://doi.org/10.1007/978-3-642-12002-2_14

[3] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.

[4] Amir M. Ben-Amram. 2010. Size-Change Termination, Monotonicity Constraints and Ranking Functions. *Logical Methods in Computer Science* 6, 3 (2010). http://arxiv.org/abs/1005.0253

[5] Amir M. Ben-Amram. 2011. Monotonicity Constraints for Termination in the Integer Domain. *Logical Methods in Computer Science* 7, 3 (2011). https://doi.org/10.2168/LMCS-7(3:4)2011

[6] Amir M. Ben-Amram and Samir Genaim. 2013. On the Linear Ranking Problem for Integer Linear-Constraint Loops. In *Proceedings of 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2429069.2429078

[7] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55. https://doi.org/10.1145/2629488

[8] Amir M. Ben-Amram and Samir Genaim. 2015. Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions. In *Proceedings of 27th International Conference on Computer Aided Verification (CAV'15)*, Vol. 9207. Springer, Cham, 304–321. https://doi.org/10.1007/978-3-319-21668-3_18

[9] Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *Proceedings of 29th International Conference on Computer Aided Verification (CAV'17)*, Vol. 10427. Springer, Cham, 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

[10] Dirk Beyer. 2017. Software Verification with Validation of Results - (Report on SV-Comp 2017). In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*. Springer-Verlag New York, Inc., New York, NY, USA, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 184–190. http://dl.acm.org/citation.cfm?id=2032305.2032321

[12] František Blahoudek, Matthias Heizmann, Sven Schewe, Jan Strejček, and Ming-Hsien Tsai. 2016. Complementing Semi-deterministic Büchi Automata. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 770–787. https://doi.org/10.1007/978-3-662-49674-9_49

[13] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, Vol. 10205. Springer, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

[14] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 491–504. https://doi.org/10.1007/11513988_48

[15] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, Vol. 9636. Springer, Berlin, Heidelberg, 387–393. https://doi.org/10.1007/978-3-662-49674-9_22

[16] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2013. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design* 43, 1 (2013), 93–120. https://doi.org/10.1007/s10703-013-0186-4

[17] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 415–426. https://doi.org/10.1145/1133981.1134029

[18] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving Program Termination. *Commun. ACM* 54, 5 (2011), 88–98. https://doi.org/10.1145/1941487.1941509

[19] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Proceedings of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

[20] Nathanaël Courant and Caterina Urban. 2017. Precise Widening Operators for Proving Termination by Abstract Interpretation. In *Proceedings of 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, Vol. 10205. Springer, Berlin, Heidelberg, 136–152. https://doi.org/10.1007/978-3-662-54577-5_8

[21] Patrick Cousot and Radhia Cousot. 2012. An Abstract Interpretation Framework for Termination. In *Proceedings of 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/2103656.2103687

[22] Jean-Michel Couvreur. 1999. On-the-fly Verification of Linear Temporal Logic. In *Proceedings of International Symposium on Formal Methods (FM'99)*. Springer-Verlag, London, UK, UK, 253–271. https://doi.org/10.1007/3-540-48119-2_16

[23] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. 2006. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proceedings of 18th International Conference on Computer Aided Verification (CAV'06)*, Vol. 4144. Springer, Berlin, Heidelberg, 17–30. https://doi.org/10.1007/11817963_5

[24] Seth Fogarty and Moshe Y. Vardi. 2012. Büchi Complementation and Size-Change Termination. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:13)2012

[25] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. 2009. Proving Termination of Integer Term Rewriting. In *Proceedings of 20th International Conference on Rewriting Techniques and Applications (RTA'09)*. Springer-Verlag, Berlin, Heidelberg, 32–47. https://doi.org/10.1007/978-3-642-02348-4_3

[26] Andreas Gaiser and Stefan Schwoon. 2009. Comparison of Algorithms for Checking Emptiness of Büchi Automata. In *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, Vol. 13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/2349

[27] Pierre Ganty and Samir Genaim. 2013. Proving Termination Starting from the End. In *Proceedings of 25th International Conference on Computer Aided Verification (CAV'13)*, Vol. 8044. Springer-Verlag New York, Inc., New York, NY, USA, 397–412. https://doi.org/10.1007/978-3-642-39799-8_27

[28] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reasoning* 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[29] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2006. Mechanizing and Improving Dependency Pairs. *J. Autom. Reasoning* 37, 3 (2006), 155–203. https://doi.org/10.1007/s10817-006-9057-7

[30] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Springer. https://doi.org/10.1007/3-540-60761-7

[31] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Proceedings of 17th International Conference on Static Analysis (SAS'10)*. Springer-Verlag, Berlin, Heidelberg, 304–319. http://dl.acm.org/citation.cfm?id=1882094.1882113

[32] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. 2013. Linear Ranking for Linear Lasso Programs. In *Proceedings of 15th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, Vol. 8172. Springer, Cham, 365–380. https://doi.org/10.1007/978-3-319-02444-8_26

[33] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Proceedings of 26th International Conference on Computer Aided Verification (CAV'14)*. Springer-Verlag New York, Inc., New York, NY, USA, 797–813. https://doi.org/10.1007/978-3-319-08867-9_53

[34] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Proceedings of 22nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 89–103. https://doi.org/10.1007/978-3-642-14295-6_9

[35] Robert P. Kurshan. 1987. Complementing Deterministic Büchi Automata in Polynomial Time. *J. Comput. Syst. Sci.* 35, 1 (1987), 59–71. https://doi.org/10.1016/0022-0000(87)90036-5

[36] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-termination Specification Inference. In *Proceedings of 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 489–498. https://doi.org/10.1145/2737924.2737993

[37] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

[38] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. 2012. Termination Analysis with Algorithmic Learning. In *Proceedings of 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 88–104. https://doi.org/10.1007/978-3-642-31424-7_12

[39] Jan Leike and Matthias Heizmann. 2015. Ranking Templates for Linear Loops. *Logical Methods in Computer Science* 11, 1 (2015). https://doi.org/10.2168/LMCS-11(1:16)2015

[40] Max Michel. 1988. *Complementation is more difficult with automata on infinite words.* Technical Report. CNET, Paris.

[41] Sebastian Ott. 2016. Implementing a Termination Analysis using Configurable Software Analysis. Master's Thesis, University of Passau, Software Systems Lab.

[42] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing Liveness to Safety in First-Order Logic. *ACM Program. Lang.* 2, POPL (2018), 26:1–26:33. https://doi.org/10.1145/3158114

[43] Doron Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *Proceedings of 5th International Conference on Computer Aided Verification (CAV'93)*. Springer-Verlag, London, UK, 409–423. http://dl.acm.org/citation.cfm?id=647762.735490

[44] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proceedings of 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, Vol. 2937. Springer, Berlin, Heidelberg, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[45] Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *Proceedings of 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. IEEE Computer Society, Washington, DC, USA, 32–41. https://doi.org/10.1109/LICS.2004.50

[46] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. 2008. Heap Assumptions on Demand. In *Proceedings of 20th International Conference on Computer Aided Verification (CAV'08)*. Springer-Verlag, Berlin, Heidelberg, 314–327. https://doi.org/10.1007/978-3-540-70545-1_31

[47] Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-threaded Programs. In *Proceedings of 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*. Springer-Verlag, Berlin, Heidelberg, 237–251. https://doi.org/10.1007/978-3-642-28756-5_17

[48] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. 2017. Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic. *J. Autom. Reasoning* 58, 1 (2017), 33–65. https://doi.org/10.1007/s10817-016-9389-x

[49] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *Proceedings of 24th International Symposium on Static Analysis (SAS'13)*, Vol. 7935. Springer, Berlin, Heidelberg, 43–62. https://doi.org/10.1007/978-3-642-38856-9_5

[50] Caterina Urban. 2015. FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution). In *Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*. Springer-Verlag New York, Inc., New York, NY, USA, 464–466. https://doi.org/10.1007/978-3-662-46681-0_46

[51] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *Proceedings of 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*. Springer-Verlag New York, Inc., New York, NY, USA, 54–70. https://doi.org/10.1007/978-3-662-49674-9_4

[52] Caterina Urban and Antoine Miné. 2014. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *Proceedings of 23rd European Symposium on Programming Languages and Systems (ESOP'14)*. Springer-Verlag New York, Inc., New York, NY, USA, 412–431. https://doi.org/10.1007/978-3-642-54833-8_22

[53] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Proceedings of 21st International Symposium on Static Analysis (SAS'14)*, Vol. 8723. Springer, Cham, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

[54] Antti Valmari. 1991. Stubborn Sets for Reduced State Space Generation. In *Proceedings of 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets (ICATPN'89)*. Springer, 491–515. https://doi.org/10.1007/3-540-53863-1_36