Masaryk University
Faculty of Informatics

Complementation of Semi-Deterministic Transition-Based Generalized Büchi Automata

Bachelor's Thesis

Adam Fiedler

Brno, Spring 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Adam Fiedler

Advisor: doc. RNDr. Jan Strejček, Ph.D.
Consultant: RNDr. František Blahoudek, Ph.D.

Acknowledgements

I would very much like to thank Jan Strejček and František Blahoudek for introducing me into this topic and for their invaluable suggestions that helped me get to the finish line. Their patience and eagerness to discuss ideas day and night were always inspiring. I am also deeply grateful to my family and Petra for their enduring support.

Abstract

The thesis presents an extension of the NCSB algorithm to complement the *semi-deterministic transition-based generalized Büchi automaton* (SDTGBA). To the best of our knowledge, this extension is the first existing algorithm that can complement the SDTGBA without transforming the input automaton. We prove the extension's correctness, analyze its complexity, and suggest several optimizations. An implementation of the extension written using the SPOT library is included and experimentally compared with two standard complementation algorithms used on the SDTGBA.

Keywords

ω-language, SDTGBA, complementation, semi-determinism, SPOT

Contents

List of Tables

List of Figures

1 Introduction

Complementation of Büchi automata has become a prominent problem, for example, in *model checking*. When program runs are represented by an ω-language (a set of infinite words over a finite alphabet), a requirement on their properties can be *modeled* as a Büchi automaton (BA). Instead of testing a set L of the program runs explicitly for inclusion $L \subseteq M$, where M is a set representing the required properties, algorithmic methods test for $L \cap \overline{M} = \emptyset$, where $\overline{M}$ is the *complement* language of M. This is done by performing a well-known *emptiness check* on the intersection of a BA accepting L and a constructed complement automaton of a BA accepting M [1, p. 261].

In stark contrast with finite automata, the construction of the complement BA is challenging because Büchi automata are generally nondeterministic while their deterministic equivalents do *not* have to exist without a more complex acceptance condition [2, p. 16]. Interestingly enough, Blahoudek et al. have observed that some model checking tools produce a more approachable class called *semi-deterministic* automata. Informally, these automata are behaving deterministically from a certain point onward, which has turned out to be very useful for complementation as shown by the *NCSB* algorithm [3]. NCSB with its recent optimization called *NCSB-Lazy* continues to be the most efficient algorithm for complementing the semi-deterministic BA (SDBA) up to date [4, p. 141].

However, practicality has brought new types of automata with Büchi-like acceptance. One such type is called the *transition-based generalized Büchi automaton* (TGBA), whose transition-based acceptance and a little more articulate acceptance condition format among other benefits usually lead to smaller automata in terms of states [5, p. 357]. Since a semi-deterministic TGBA (SDTGBA) can be transformed into a conventional SDBA using *degeneralization* [5], the straightforward idea is to complement an SDTGBA just by applying NCSB after degeneralization. Imaginably though, degeneralizing automata can produce redundancies, some of which might remain even after applying state space reduction algorithms. The thesis aims to present the first *direct* way to complement the SDTGBA and save states by skipping degeneralization.

Building upon the foundations laid out by NCSB, it turns out a theoretical extension working directly with the SDTGBA can be defined without *any* transformations of the input automaton. To properly evaluate this extension, we needed an implementation preferably written with the help of a library containing other algorithms available for performance comparison, primarily NCSB itself to be run on the transformed SDTGBA to the SDBA. Hence we have chosen *SPOT* [6], the ω-automata tool containing usable algorithms for various manipulations besides complementation and degeneralization. To name one, SPOT contains a number of state space reduction algorithms, which we call *SPOT's optimizations*. Performance evaluation was made both with and without applying these optimizations. In both cases, the extension seems to outperform NCSB running on degeneralized automata. Another advantage is that the extension generates a transition-based Büchi automaton with only one acceptance set, which works the same as the conventional (transition-based) BA.

The thesis starts with formalizing the concepts of the TGBA, complementation, and semi-determinism (Chapter 2). A helpful collection of formal tools used throughout the text is also introduced. This is followed by a brief overview of the original NCSB for a better understanding of the decisions made for the extension design. In Chapter 3, the extension itself is formally defined, followed by a complexity analysis and a proof of correctness. We conclude the chapter with several suggested optimizations. The rest of the work (Chapter 4) deals with implementing the extension in the SPOT library, describing the experimental evaluation and presenting results thereof (namely Sections 4.1, 4.2).

There is still great room for improvement outside the scope of this thesis. The evaluation has shown the extension performs *worse* when compared with the SPOT's default tool for complementation called `autfilt --complement`. This requires further analysis, although at least one new direction for additional optimizations exists as suggested by *NCSB-Lazy* [4, p. 143]. Despite the results, we hope that the presented ideas can eventually be used for cheaper complementation of the SDTGBA in SPOT.

2 Preliminaries

By a *transition-based generalized Büchi automaton* (TGBA) we mean a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ where

- Q is a finite set of *states*,
- Σ is a finite *alphabet*,
- $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*,
- $q_I \in Q$ is an *initial state*,
- $\mathcal{F} = \{F_0, \dots, F_k\} \subseteq \mathcal{P}(\delta)$ is a non-empty set of *acceptance sets* containing *accepting transitions*.

Any triple $(q_s, a, q_d) \in \delta$ is called a *transition* from q_s under a to q_d, where q_s is called the *source* and q_s is called the *destination* of the transition. A *run* of $\mathcal{A}$ on an infinite word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$ is any maximal sequence of *adjoining* transitions[1]

$$\rho = (q_0, w_0, q_1)(q_1, w_1, q_2) \ldots (q_{i-1}, w_{i-1}, q_i)(q_i, w_i, q_{i+1}) \ldots \in \delta^+ \cup \delta^\omega$$

such that $q_0 = q_I$, where q_j is called the *successor* of q_i for *any* $j \geq i$.

If a run ρ is finite, then we say ρ *halts*. Any subsequence of ρ is said to *contain* a state $q \in Q$ if q is the source or the destination of a transition in this subsequence. Using this definition, q is *unreachable* if there does not exist w such that a run on w contains q. Furthermore, a run ρ_1 of $\mathcal{A}$ on w is said to be *intersecting* another run $\rho_2 \neq \rho_1$ of $\mathcal{A}$ on w when ρ_1 and ρ_2 share a common non-empty suffix.

We say ρ is *accepting* if and only if it contains infinitely many transitions from each $F_i \in \mathcal{F}$, *non-accepting* otherwise. Subsequently, w is *accepted* by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on w. The *ω-language* of $\mathcal{A}$ denoted $L(\mathcal{A})$ is the set of all the infinite words accepted by $\mathcal{A}$. If for each $w \in L(\mathcal{A})$ there exists only one accepting run, then $\mathcal{A}$ is called *unambiguous*.

1. Maximal means that if a run is finite, then the run's last destination is not a source of any transition under the next letter.

By a *complement automaton* $\mathcal{C}$ of $\mathcal{A}$ it is meant such TGBA that $L(\mathcal{C}) = \Sigma^\omega \setminus L(\mathcal{A})$, hence *complementation* of $\mathcal{A}$ just stands for constructing such a $\mathcal{C}$.

To stay concise, we *always* assume that

$$\bigcup \mathcal{F} = \bigcup_{F_i \in \mathcal{F}} F_i.$$

Let us also introduce some special notation for the TGBA. Firstly, $q_1 \xrightarrow{a} q_2$ shall denote *any* triple (q_1, a, q_2). We do *not* restrict these triples to δ because we make great use of this notation on transitions of both the input automaton and its complement to be constructed, even on subsets of transitions as apparent in the following definition.

Since δ is defined as a relation for technical reasons, we include a (total) function $\Delta : Q \times \Sigma \times \mathcal{P}(\delta) \to \mathcal{P}(Q)$ that returns destinations. It is defined as

$$\Delta(q_1, a, T) \overset{\text{def}}{=} \{q_2 \mid q_1 \xrightarrow{a} q_2 \in T\},$$

and further extended to sets $X \subseteq Q$ as

$$\Delta(X, a, T) \overset{\text{def}}{=} \bigcup_{q \in X} \Delta(q, a, T).$$

Here T can be understood as a *template* to build Δ for a given (sub)set of transitions. For example, $\Delta(X, a, \bigcup \mathcal{F})$ returns exactly destinations of accepting transitions from q under a. A trivial consequence is that there is an accepting transition under a in $F_i \in \mathcal{F}$ from a state in X if and only if $\Delta(X, a, F_i) \neq \emptyset$. For simplification, we just write $\Delta(q, a)$ and $\Delta(X, a)$ when referring to $\Delta(q, a, \delta)$, $\Delta(X, a, \delta)$ respectively.

2.1 Semi-Deterministic TGBA (SDTGBA)

Before looking into semi-determinism, let us first briefly review what we mean by determinism. A state $q \in Q$ is called *deterministic* if and only if $|\Delta(q, a)| \leq 1$ for all $a \in \Sigma$, *non-deterministic* otherwise. We call $\mathcal{A}$ deterministic (DTGBA) when each *reachable* state $q' \in Q$ is deterministic.

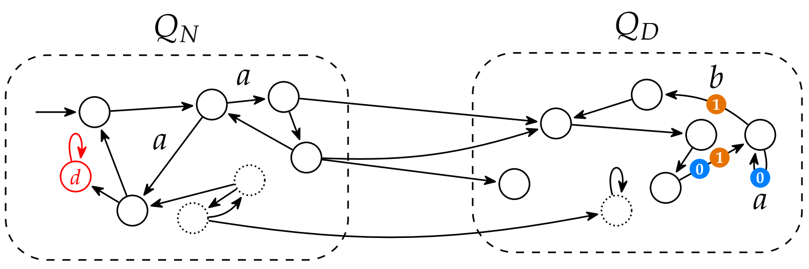


Figure 2.1: A semi-deterministic TGBA $\mathcal{A}$ with $\Sigma = \{a, b\}, \mathcal{F} = \{F_0, F_1\}$. Dashed states are unreachable, the state d in Q_N could be moved to Q_D if chosen. Marked transitions in Q_D belong to their corresponding acceptance sets: for example, the transition marked ⓪ belongs to F_0, the transition marked ⓪ ① belongs to *both* F_0 and F_1 (the colors have no special meaning).

The thesis presents an approach to complementation of a special class of the TGBA called the *semi-deterministic* TGBA (SDTGBA). Formally, a TGBA $\mathcal{A}$ is semi-deterministic when $Q = Q_N \cup Q_D$ such that Q_N, Q_D are disjoint and for each $a \in \Sigma$ these three conditions are met.

SD1) $\Delta(Q_D, a) \subseteq Q_D$

(there is no transition from a state in Q_D to a state in Q_N),

SD2) $\forall q \in Q_D : |\Delta(q, a)| \leq 1$

(states in Q_D are deterministic),

SD3) $\forall F_i \in \mathcal{F} : F_i \subseteq (Q_D \times \Sigma \times Q_D)$

(accepting transitions exist only between the states in Q_D).

We sometimes refer to Q_N as the *non-deterministic part* and to Q_D as the *deterministic part* of an SDTGBA $\mathcal{A}$. The transitions in $(Q_N \times \Sigma \times Q_D) \cap \delta$ are called *transit transitions*.

An example of an SDTGBA is sketched in Figure 2.1, although we have to note that the definition is flexible enough to allow many deviations from an expectable setup, such as having the initial state in Q_D (rendering the automaton deterministic) or each state in Q_D unreachable (thus $L(\mathcal{A}) = \emptyset$). This needs to be accounted for in a correct complementation algorithm for the SDTGBA. We build our algorithm upon the principles of an earlier work called NCSB [3].

2.2 Intuition into the NCSB Algorithm

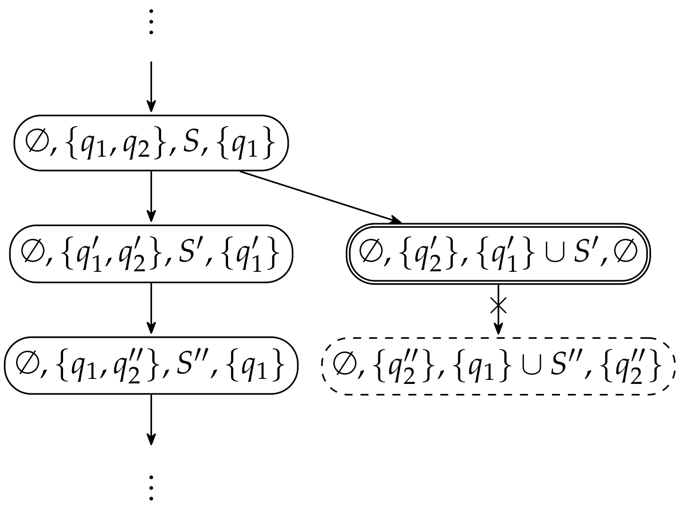
NCSB is a *guess-and-check* complementation algorithm introduced by Blahoudek et al. [3] for the conventional state-based semi-deterministic Büchi automaton (SDBA). We leave out the definition of the SDBA because its exact form is not essential for the upcoming informal presentation. The reader can imagine an SDBA as an SDTGBA we have defined in Section 2.1 with the exceptions that acceptance is based on *accepting states* ($\mathcal{F} \subseteq \mathcal{P}(Q)$) instead of accepting transitions and that there is only *one* such set of accepting states ($|\mathcal{F}| = 1$).

One of the options how to decide whether an infinite word w belongs to the complement ω-language of an SDBA is verifying that every run of the SDBA on w contains finitely many accepting states [3, p. 774]. That is why NCSB keeps track of *all* the runs: it analyzes run trees, an occurrence common to several other complementation algorithms [1].

The theory behind NCSB works with a dynamic property of runs which is termed *safety*. A run is said to *become* safe once after reading some finite prefix of w it contains no more accepting states (i.e., the remaining suffix of the run does not contain any accepting states) [3, p. 774]. A trivial consequence of the definition then is that a safe run cannot become unsafe again.

We do not know in advance *when* a particular run becomes safe, by when meaning what prefix of w it takes for the run to stop visiting accepting states. Here *guessing* comes into play: NCSB at times non-deterministically guesses if a run has already become safe or not [3, p. 775]. A way to understand it is that a guess splits the considered path in the run tree of NCSB on w into at least two branches: one for the positive and one for the negative answer. The negative branch naturally behaves as if nothing has changed. The positive one, however, starts *checking* if the run guessed to be safe is really safe. Note that this is possible thanks to the fact that accepting states are *deterministic*.

If a run of the input automaton is to visit an accepting state in the positive branch after all, the branch is cut off (by not defining a transition to such a configuration), rendering the complement run made up of this path non-accepting as it halts (see Figure 2.2). However, if the checking is infinitely successful for the remaining suffix of w, the guess must have been correct. If there exists a path in the run tree of



$\emptyset, \{q_1, q_2\}, S, \{q_1\}$
$\emptyset, \{q_1', q_2'\}, S', \{q_1'\}$
$\emptyset, \{q_2'\}, \{q_1'\} \cup S', \emptyset$
$\emptyset, \{q_1, q_2''\}, S'', \{q_1\}$
$\emptyset, \{q_2''\}, \{q_1\} \cup S'', \{q_2''\}$

Figure 2.2: NCSB incorrectly guessing safety of a run that contains $q_1 \in F_0$ (successors of q_2 are not accepting).

NCSB which for each run of the input automaton eventually guesses correctly that it became safe, it follows indeed there is no accepting run of the input automaton on w because an accepting run *cannot* ever become safe.

NCSB uses several powerful concepts to put the theory into practice. The fundamental concept is state representation of runs. Because there are a possibly infinite amount of runs, they need a finite representation. That is why an automaton constructed by NCSB works with four sets of states in which a state $q \in Q$ represents all runs that have reached q after reading a finite prefix of w.

The set introduced as N contains all runs which have not yet left the non-deterministic part of the input automaton (hence N), which technically means N is a subset of non-deterministic states. However, it is also an implementation of a well-known concept named a *powerset construction* [3, p. 775]. N can grow because it computes all non-deterministic successors of the contained states, it is a union of sets of successors.

The *safe set* S contains only runs that should be safe; this practically translates so that it cannot contain any accepting states. If a run in S

reaches an accepting state, some earlier guess was incorrect. NCSB defines no transition to a configuration of S containing an accepting state, forcing any complement run making the wrong guess to halt.

The set C contains runs which have reached the deterministic part but have not been moved to S yet. It computes the determined successors of contained states and accepts new runs in the deterministic part (i.e., those which have just reached the deterministic part). If an accepting state appears in C, its successor can non-deterministically stay in C or be moved to S. This represents a guess whether the accepting state was the last one contained in the corresponding run.

The *breakpoint* set $B \subseteq C$ is a variation of another technique called a *breakpoint construction*. In NCSB, B almost copies the behavior of C with the difference that when non-empty, B *does not* accept new runs in the deterministic part. When a subset of runs contained in B is moved to S, it follows B must shrink as a consequence. Shrinking repeats until $B = \emptyset$, which the authors call reaching a *breakpoint* [3, p. 775], and new runs from C are copied to B so that the process can start again.

B can be interpreted as a magnifying glass on a subtree of runs in the deterministic part, making sure they all eventually leave for S, then moving on to check an adjoining subtree. Without B, we speculate that NCSB would have hardly another way how to define its acceptance condition, which needs to ensure *all* runs are being checked from a certain point. C alone behaves too unpredictably to determine in the accepting formalism that a subset of runs in C have left for S. On the other hand, B makes it trivial: if it is *emptied* infinitely many times, then we know that no run avoided checking by staying in C.

The complementation itself is computing correct configurations of these sets and correct transitions between them. States of the complement automaton are instances of quadruples (N, C, S, B), and those which contain an empty breakpoint are accepting. Transitions are computed as informally described in the previous paragraphs when saying what these sets *compute* or *do*. The mentioned principles stand behind what NCSB does at its core and we shall make great use of them. All will hopefully become clear once our NCSB extension to complement the SDTGBA is formally defined.

3 SDTGBA Complementation

Unlike the SDBA, the SDTGBA use *transitions* to define their acceptance condition (hence *transition-based*) and we need to take this into consideration when creating a complement construction for them. However, it is a problem only of technical importance and should not present a great difficulty.

The real challenge for the extension to accept a word w is checking whether every run of the input automaton on w eventually stops visiting at least one of the acceptance sets (when it becomes safe *relative* to this set), because for no run we know in advance which acceptance set it will be (it can vary).

As in the case of NCSB, we also use the *guess-and-check* strategy and state representation of runs using the familiar sets of states. The set N is used to track runs in the non-deterministic part of the SDTGBA, the set C tracks still possibly unsafe runs in the deterministic part and the set B ensures every run is eventually checked. These sets are explained in Section 2.2.

The difference in our approach is that we have *several* safe sets $S_0, \dots, S_k$ where each S_i contains runs which should be safe relative to its *corresponding* acceptance set F_i. Intuitively, this means the extension tries to guess non-deterministically for every run the correct safe set, checking is then analogous to NCSB. If a run in S_i uses a transition from F_i, the guess was incorrect and the complement run that made this guess halts to prevent accepting a possibly incorrect word.

To minimize the number of guesses, the extension restricts the moments when a particular run can be non-deterministically moved to one of $S_0, \dots, S_k$. This is only when the run uses an accepting transition, or when the run enters the deterministic part (because some runs can contain no accepting transitions at all). The type of the used accepting transition also gives the non-deterministic choices for the safe sets (which is described in detail under the formal definition).

Moreover, such an optimization means that the only occurrence of non-determinism will be exactly at these two mentioned places (described again under the formal definition). Further non-trivial optimizations are placed at the end of the chapter because they come at a price of increasing technical difficulty.

3.1 Formal Construction

Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ be an SDTGBA where $Q = Q_N \cup Q_D$ such that the conditions for semi-determinism are met and $\mathcal{F} = \{F_0, \dots, F_k\}$. We define the complement automaton $\mathcal{C} = (R, \Sigma, \delta_{\mathcal{C}}, r_I, \{F_{\mathcal{C}}\})$ of $\mathcal{A}$ as follows.

- $R \subseteq \mathcal{P}(Q_N) \times \mathcal{P}(Q_D) \times \mathcal{P}(Q_D)^{k+1} \times \mathcal{P}(Q_D)$,
- $r_I = (\{q_I\}, \emptyset, \underbrace{\emptyset, \dots, \emptyset}_{\text{safe sets}}, \emptyset)$,
- $F_{\mathcal{C}} = \{r \xrightarrow{a} (N', C', S'_0, \dots, S'_k, B') \in \delta_{\mathcal{C}} \mid B' = \emptyset\}$,
- and the transition

  $$(N, C, S_0, \dots, S_k, B) \xrightarrow{a} (N', C', S'_0, \dots, S'_k, B') \in \delta_{\mathcal{C}}$$

  if and only if

  E1) $N' = \Delta(N, a) \cap Q_N$
  (tracking the runs in the non-deterministic part correctly),

  E2) $C' \cup S'_0 \cup \dots \cup S'_k$
  $= \Delta(C \cup S_0 \cup \dots \cup S_k, a) \cup \big(\Delta(N, a) \cap Q_D\big)$
  (tracking the runs in the deterministic part correctly),

  E3) $C' \supseteq \Delta(C, a, \delta \setminus \bigcup \mathcal{F})$
  (only runs using an accepting transition can leave C to a safe set),

  E4) $C' \cap (S'_0 \cup \dots \cup S'_k) = \emptyset$
  (runs cannot be safe and unsafe at the same time),

  E5) $S'_i \cap S'_j = \emptyset$ for each $0 \leq i < j \leq k$
  (runs need to be safe only in one of the safe sets),

  E6) $\Delta(q, a) \neq \emptyset$ for each $q \in C$
  (*finite* runs should be moved to one of the safe sets before halting),

E7) if $B \neq \emptyset$ then $B' = \Delta(B, a) \cap C'$ else $B' = C'$
(*B* tracks a fixed subset of runs staying in *C* until a breakpoint is reached, then it is filled with new runs in *C*),

E8) $S'_i \subseteq \Delta(S_i, a) \cup \Delta(C, a, F_i) \cup \big(\Delta(N, a) \cap Q_D\big)$
for each $0 \leq i \leq k$
(possible additions to S_i are given by accepting transitions from F_i used by runs in *C* and transit transitions used by runs entering Q_D),

E9) $S'_i \supseteq \Delta(S_i, a)$ for each $0 \leq i \leq k$
(safe runs stay safe in the same safe set),

E10) $\Delta(S_i, a, F_i) = \emptyset$ for each $0 \leq i \leq k$
(safe runs cannot use a corresponding accepting transition).

As in the case of NCSB, note that any $r = (N, C, S_0, \ldots, S_k, B) \in R$ is deterministic except for the following two cases, which are important for proving Theorem 2.

- A run represented by q_2 immediately after using a transit transition ($q_2 \in \Delta(N, a) \cap Q_D$ and a has just been read) can either be left in C' or moved into one of the safe sets $S'_0, \ldots, S'_k$.
- A run represented by q immediately after an accepting transition from both F_i and F_j ($q \in \Delta(C, a, F_i \cap F_j)$ and a has just been read) can either be left in C' or moved into one of S'_i or S'_j.

On the other hand, $\mathcal{C}$ is *not* unambiguous opposed to NCSB constructions. This is for two primary reasons, which are not trivial to circumvent.

- A run can be safe relative to *several* acceptance sets and $\mathcal{C}$ as defined above does not enforce a particular choice.
- A run can either take its own opportunity to be moved to one of the safe sets S_i (that is when using a transit transition or immediately after using an accepting transition from F_i), or in certain cases wait for an *intersecting* run to be moved to S_i together (see Figure 3.1).

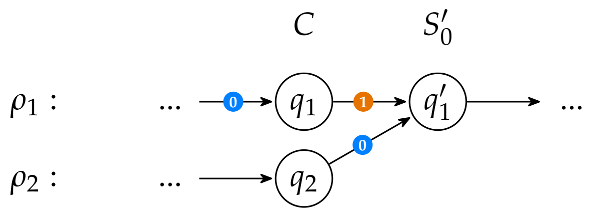


Figure 3.1: A delayed jump to S_0 with an intersecting run. Marked transitions belong to the corresponding acceptance sets $F_0, F_1 \in \mathcal{F}$. If the transition marked ❶ was not accepting, the delayed jump would be forbidden by E3.

In the following text, we shall strictly refer to runs of the complement automaton $\mathcal{C}$ as *superruns* (even though they are TGBA runs in the ordinary sense) and to the states of $\mathcal{C}$ as *superstates* (even though they are TGBA states in the ordinary sense). This is to prevent any misunderstandings connected with interchangeability.

3.2 Complexity

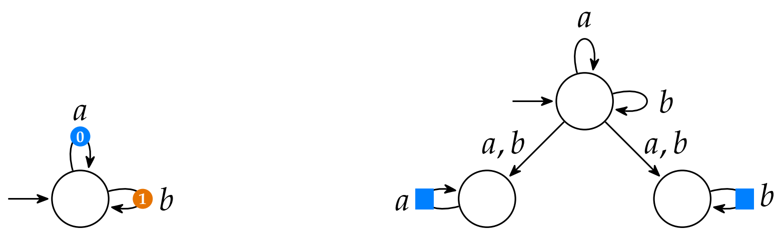
The superstate space is strikingly similar to that of NCSB. However, unlike NCSB which forbids accepting states in S, the extension places no general restrictions for membership in the sets of its tuples other than members' (non-)deterministic behavior and the fact that all sets except C and B are disjoint while $B \subseteq C$.

This can surely be improved as discussed in Section 3.4.3. For now, it does not prevent us from deducing the upper bound on the states of the constructed complement. Let us consider any superstate $r = (N, C, S_0, \ldots, S_k, B) \in R$, then

- a state $q \in Q_N$ either is or is not present in N, and
- a state $q \in Q_D$ is either in just C or in both C and B or in just one of the $k+1$ sets $S_0, \ldots, S_k$ (because they are mutually disjoint) or not present in r at all.

Thus the upper bound is apparently given by

$$|R| \leq 2^{|Q_N|} \cdot (3 + (k+1))^{|Q_D|}.$$



(a) The input DTGBA $\mathcal{A}$ (b) The complement of $\mathcal{A}$

Figure 3.2: Standard complementation of a total DTGBA with $\mathcal{F} = \{F_0, F_1\}$, where marked transitions belong to the corresponding acceptance sets. The transitions marked ■ belong to the only acceptance set of the complement automaton.

In conclusion of this short analysis, we must mention a nice property that since a DTGBA is semi-deterministic with $q_I \in Q_D$, its complement can be constructed using our extension. Because there always exists only one run of a DTGBA on a word w, we know in addition to the aforementioned that

- $max\{|N|, |C|, |S_0|, \ldots, |S_k|, |B|\} \leq 1$,
- $C = B$, and
- $N \neq \emptyset$ only in r_I (when $q_I \in N$ and $C = S_0 = \ldots = S_k = \emptyset$).

Hence for each $r \neq r_I$, there exists a single state $q \in Q_D$ which is either in $C = B$ or in exactly one of the $k + 1$ safe sets $S_0, \ldots, S_k$ if we assume the input DTGBA to be total. The upper bound on the number of states for our construction used on a total DTGBA is then

$$|R| \leq 1 + \big(1 + (k+1)\big) \cdot |Q_D|.$$

This is analogous to the standard complementation algorithm for the total DTGBA (see Figure 3.2), whose upper bound on the number of states is $(1 + (k+1)) \cdot |Q_D|$. It works by making a copy of the input DTGBA for each acceptance set and then cutting off corresponding accepting transitions in each copy, while only the remaining transitions in the copies are accepting. A run can jump from the original automaton non-deterministically to one of the copies (guessing it will use no more corresponding accepting transitions); only there it can become accepting.

3.3 Correctness

To avoid redundancy in the following text, we often omit information unnecessary for our argument when discussing transitions of a superrun. At least these phrases are used repetitively throughout the proofs.

- By saying that the superrun *contains a transition to* $x \in N$ (under $a \in \Sigma$), we mean the superrun contains a transition

  $$q \xrightarrow{a} (N, C, S_0, \dots, S_k, B) \in \delta_C$$

  such that $x \in N$, with an analogous meaning for transitions to $x \in C$, to $x \in S_0, \dots, S_k$, or to $x \in B$.

- When the superrun *contains a transition to* $B = \emptyset$ (under a), this means it contains a transition $q \xrightarrow{a} (N, C, S_0, \dots, S_k, \emptyset) \in \delta_C$.

- The meaning is in both of the previous cases symmetrical when we speak of the superrun containing a transition (under a) *from* $B = \emptyset$, from $x \in N$, $x \in C$, etc.

- We plentifully use the defined term *successor* of a state $q_i \in Q$. Please recall its definition, because it always refers to *any* successor of q_i in a run (usually given by context), including q_i itself: q_i is a successor of q_i for technical reasons. We *never* use the term for superstates.

Theorem 1. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ be an SDTGBA where $Q = Q_N \cup Q_D$ such that the conditions for semi-determinism are met and $\mathcal{C}$ be the complement automaton of $\mathcal{A}$ as defined in Section 3.1. Then*

$$\forall w = w_0 w_1 w_2 w_3 \ldots \in \Sigma^\omega : w \in L(\mathcal{A}) \implies w \notin L(\mathcal{C}).$$

Proof. Let ρ be an accepting run of $\mathcal{A}$ on w. By definition ρ contains infinitely many transitions from each $F_i \in \mathcal{F}$. This also means that there is a suffix of ρ containing only states in Q_D because $\mathcal{A}$ is semi-deterministic (SD1, SD3). Let ρ_D be such a suffix that

$$\rho = \ldots (q_i \xrightarrow{w_i} q_{i+1}) \underbrace{(q_{i+1} \xrightarrow{w_{i+1}} q_{i+2}) \ldots}_{\rho_D}$$

where either it holds that $q_i = q_0$ or $q_i \in Q_N$, and that $q_{i+1} \in Q_D$.

Suppose for a contradiction that $w \in L(\mathcal{C})$. For an accepting and thus infinite superrun ζ of $\mathcal{C}$ on w, it is trivial to see by E1 that ζ contains a transition from $q_i \in N$. Now w_i is read and hence by E2 ζ must also contain a transition to $q_{i+1} \in C \cup S_0 \cup \ldots \cup S_k$. By E4, two mutually exclusive possibilities arise here:

P1) $q_{i+1} \in S_0 \cup \ldots \cup S_k$. Thus q_{i+1} is by E5 only in one of the safe sets, let us call it S_j. Hence by E9, for each successor q_l of q_{i+1} contained in ρ_D, ζ contains a transition to $q_l \in S_j$. Therefore the infinite ζ especially contains a transition to such $q_l \in S_j$ that $q_l \xrightarrow{w_l} q_{l+1} \in F_j$, since ρ_D contains infinitely many transitions from each $F_0, \ldots, F_k$ by assumption. This is in contradiction with E10.

P2) $q_{i+1} \in C$. If $q_{i+1} \in B$, after finitely many transitions there must follow a transition to $B = \emptyset$ by ζ being accepting. Because ρ is infinite, this must also mean by E2 and E7 that there is a successor q_m of q_{i+1} such that ζ contains a transition to $q_m \in S_0 \cup \ldots \cup S_k$, for which we argue in the same way as for q_{i+1} in P1.

If $q_{i+1} \notin B$, after finitely many transitions there must follow a transition to $B = \emptyset$ by ζ being accepting. Because ρ is infinite, this must also mean there is a successor q_b of q_{i+1} such that ζ contains a transition to either $q_b \in B$ by E2, E3, E7 for which we argue in the same way as for q_{i+1} in the previous paragraph, or to $q_b \in S_0 \cup \ldots \cup S_k$ (because successors stay in C unless moved to S_i by E2, E3, E8) for which we argue in the same way as for q_{i+1} in P1.

□

To prove the other implication, let us consider the following. If a non-accepting run contains finitely many transitions from several sets $F_i, \ldots, F_j$, then obviously one of them must be the set with the minimal index. If we know the run to be non-accepting, then there surely exists a function returning this minimal index (see Definition 1).

What is more, for every pair of intersecting runs with all common suffixes containing only states in Q_D, it must hold that their corresponding safe set S_j with the minimal index is *the same* because their common future determines for both whether they contain finitely many transitions from the same F_j (see Lemma 1).

Definition 1. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \{F_0, \dots, F_k\})$ be an SDTGBA where $Q = Q_N \cup Q_D$ such that the conditions for semi-determinism are met. If there exists a non-accepting run ρ of $\mathcal{A}$ on w, then $min_w : Q_D \times \mathbb{N} \to \{0, \dots, k\}$ is defined such that $min_w(q, i) = j$ for each ρ whose i-th transition is from q where j is the minimal index such that ρ contains finitely many transitions from F_j.*

Lemma 1. *For all successors q' of q holds that if $min_w(q, i)$ is defined and the transition from q' is i'-th, then $min_w(q, i) = min_w(q', i')$.*

Because we know that each run is non-accepting, we must know after what prefix of w it contains no further transitions from the acceptance set (it is safe to) with the minimal index (see Definition 2). Once it becomes safe, it is always safe (see Lemma 2).

Definition 2. *If $min_w(q, i)$ is defined, then $min\_safe_w : Q_D \times \mathbb{N} \to \{0, 1\}$ is defined such that*

$$min\_safe_w(q,i) = \begin{cases} 1 & \textit{every run of } \mathcal{A} \textit{ on } w \textit{ whose i-th transition} \\ & \textit{is from } q \textit{ contains no transitions from } F_{min_w(q,i)} \\ & \textit{in its suffix starting with the i-th transition,} \\ 0 & \textit{otherwise.} \end{cases}$$

Lemma 2. *For all successors q' of q holds that if the transition from q' is i'-th then $min\_safe_w(q, i) = 1 \implies min\_safe_w(q', i') = 1$.*

Theorem 2. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ be an SDTGBA where $Q = Q_N \cup Q_D$ such that the conditions for semi-determinism are met and $\mathcal{C}$ be the complement automaton of $\mathcal{A}$ as defined in Section 3.1. Then*

$$\forall w = w_0 w_1 w_2 w_3 \ldots \in \Sigma^\omega : w \notin L(\mathcal{C}) \implies w \in L(\mathcal{A}).$$

Proof. We shall prove by contraposition. Suppose that $w \notin L(\mathcal{A})$, therefore every run of $\mathcal{A}$ on w contains finitely many transitions from some $F_j \in \mathcal{F}$ and thus for all reachable $q \in Q_D$ exists $i \in \mathbb{N}$ such that $min_w(q,i)$ is defined. For simplicity, let us set $min_w = min$ and $min\_safe_w = min\_safe$.

Consider now the superrun

$$\begin{aligned}\zeta = (\{q_I\}, \emptyset, \emptyset, \ldots, \emptyset, \emptyset) &\xrightarrow{w_0} (N^1, C^1, S_0^1, \ldots, S_k^1, B^1)\\ &(N^1, C^1, S_0^1, \ldots, S_k^1, B^1) \xrightarrow{w_1} (N^2, C^2, S_0^2, \ldots, S_k^2, B^2) \ldots\\ &(N^i, C^i, S_0^i, \ldots, S_k^i, B^i) \xrightarrow{w_i} (N^{i+1}, C^{i+1}, S_0^{i+1}, \ldots, S_k^{i+1}, B^{i+1}) \ldots\end{aligned}$$

of $\mathcal{C}$ on w (where for $A \subseteq Q$ the symbol A^i denotes the set's i-th iteration). Recall that ζ is fully *determined* except for the only two places of non-determinism we have discussed in Section 3.1. Therefore, we specify a restriction for them here. We select ζ such that

$$\begin{aligned}\forall i \in \mathbb{N} : \Big(q \in \Delta(S_m^i, w_i) \cup \Delta(C^i, w_i, F_m) \cup \big(\Delta(N^i, w_i) \cap Q_D\big)\\ \wedge\, m = min(q, i+1) \wedge min\_safe(q, i+1) = 1\Big)\\ \iff q \in S_m^{i+1}.\end{aligned}$$

Hence ζ places a run into S_i if and only if there is no $S_{i'}$ for $i' < i$ such that the run contains finitely many transitions from $S_{i'}$. The placement occurs at the first correct opportunity: either after the run uses the last accepting transition from F_i if it contains such an accepting transition, or otherwise after it uses a transit transition.

A consequence of this property that we use repetitively throughout the proof is the following. If $min\_safe(q,i) = 1$, then q is a successor of q' such that $q' \in S_{min(q,i)}^{i'}$ for some $i' \leq i$ in ζ (of which proof we leave for the reader). Let us now show that ζ is infinite, and then show ζ is accepting.

- ζ is infinite. E1, E2, E3, E7, E8 hold for all $i \in \mathbb{N}$ in ζ trivially. We have to show that the restriction does not break the remaining conditions associated with the safe sets by induction.

  Obviously the remaining conditions work for $i = 0$ and $i = 1$, hence the first transition exists. Let us assume they hold for all

$i' < i + 1$ and show that then they hold for the $i + 1$-th iteration of sets (hence the i-th transition exists).

E5: Assuming for a contradiction $q \in S_j^{i+1} \cap S_{j'}^{i+1}$ for $j \neq j'$, then from the definition of ζ, it follows that $j = min(q, i + 1) \neq min(q, i + 1) = j'$ which is a contradiction.

E9: If $q \in \Delta(S_{j'}^i, w_i)$, then we know by the definition of Δ that q is a successor of $q' \in S_{j'}^i$, which means by the definition of ζ that $j = min(q', i)$ and $min\_safe(q', i) = 1$. Since Lemma 1 and Lemma 2 holds, from the definition of ζ follows $q \in S_j^{i+1}$.

E10: If $q \in S_j^{i+1}$, then $min\_safe(q, i + 1) = 1$ and $j = min(q, i + 1)$ by the definition of ζ and thus q cannot be a source of a transition from F_j.

E4: Assuming for a contradiction $q \in S_j^{i+1} \cap C^{i+1}$, from the definition of ζ and E3 this means $q \in \Delta(C^i, w_i, \delta \setminus F_{min(q,i+1)})$, hence q is a successor of some $q_i \in C^i$. But recall it also means that $min\_safe(q, i + 1) = 1$, hence q_i is a successor of q'_i such that $q'_i \in S_j^{i'}$ for some $i' \leq i$. By E9 we know that $q_i \in S_j^i$ and thus $q_i \notin C^i$ by I.H. because $i' \leq i$, which is a contradiction.

E6: If $\Delta(q, w_i) = \emptyset$, then surely $min\_safe(q, i) = 1$. But recall this means that q is a successor of q' such that $q' \in S_{min(q,i)}^{i'}$ for some $i' \leq i$, therefore $q \in S_{min(q,i)}^i$ by E9 and thus $q \notin C^i$ by E4.

- ζ is accepting. If $q \in B^f$ for some $f \in \mathbb{N}$, then q is by the definition of ζ such that its successor q' is the destination of a last accepting transition from F_j moved into $S_j^{f'}$ for some $f' > f$, hence by E4 $q' \notin B^{f'}$ and so $|B^{f'}| < |B^f|$ by E7. However, since ζ is infinite and this applies to any non-empty B^f, especially if $|B^f| = 1$, ζ contains infinitely many transitions to $B^{f'} = \emptyset$.

$\square$

3.4 Further Optimizations

The practical problems of the extension defined in Section 3.1 are fourfold.

1. We are not making use of the transition-based acceptance even though it can saves us a few states: $\mathcal{F}_C$ can be reformulated so that empty breakpoints are skipped when possible.

2. The non-determinism that gives runs entering Q_D a choice to move to *any* one of the safe sets is so strong that it creates a lot of redundancies, which we shall explain before removing it.

3. The safe sets do not place any restrictions upon the states they contain, even though the states can be sources only of *accepting transitions* forbidden in the sets.

These problems are not handled in the original extension because they would complicate its basic idea with technical details. Another reason is that since we can already assume correctness of our extension, it is easier to show that the optimizations preserve correctness than to prove the optimized extension from scratch.

That is why they are dealt with now in the following paragraphs, building step by step the optimized construction on which the experimentally better implementation is based. The optimizations come in (logical) series: one optimization always builds upon the changes defined by the previous one. A final review of the optimized construction with all suggested optimizations is in Section 3.4.4.

3.4.1 Skipping Empty Breakpoints When Possible

If an emptied B is to be filled with new states from C, then it does not have to be emptied at all. We can *peek* to check B *would be* emptied in the next step, fill it with the new states from C right away and make such a transition accepting. Formally this can be implemented by changing the rule E7 to

E7) if $\Delta(B,a) \cap C' \neq \emptyset$ then $B' = \Delta(B,a) \cap C'$ else $B' = C'$, and then by changing the acceptance condition of the construction to

$$\begin{aligned}\mathcal{F}_C = \{&(N, C, S_0, \dots, S_k, B) \xrightarrow{a} (N', C', S'_0, \dots, S'_k, B') \in \delta_C \\ &\mid \Delta(B, a) \cap C' = \emptyset\}.\end{aligned}$$

Notice how we use both B and C' when choosing the right accepting transitions for $\mathcal{F}_C$. This is possible only thanks to the fact that unlike state-based acceptance, transition-based acceptance provides us with extra information about the previous state. The optimization preserves correctness because the original extension actually accepts every time $\Delta(B, a) \cap C' = \emptyset$, it just delays the refilling of B.

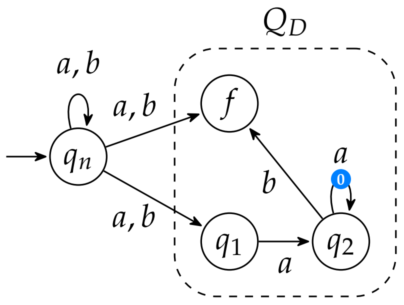
3.4.2 Reducing Non-Determinism for Runs Entering Q_D

When considered carefully, the non-determinism used on runs entering Q_D, which allows them to be moved to any one of the safe sets, generates at least $k + 1$ copies of each reachable state in the deterministic part. This is because the reachable destinations of transit transitions can be moved into any one of the sets $S_0, \dots, S_k$. Without removing it, there is not a good chance to outperform NCSB running on degeneralized automata.

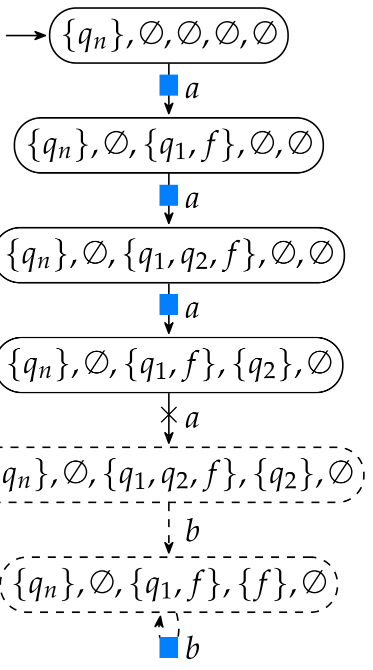
However, it cannot be removed without changing several rules of the extension because this non-determinism has a substantial task to deal with the runs not containing any accepting transitions at all. What is feasible is to decide non-deterministically whether we move runs entering Q_D or those using an accepting tranisition into just S_0 or simply leave them to C. Such an idea surely solves runs containing no accepting transitions at all and finitely many transitions from F_0.

The method to handle the remaining runs is to allow jumping through successive safe sets with S_k serving as a ceiling. The first time a run uses an accepting transition from F_i in S_i, it is automatically *moved* into the next S_{i+1} if $i < k$. If there is no available next safe set ($i = k$), then a superrun halts.

One consequence of this change needs to be fixed though. Intersecting runs can now appear in different safe sets (consider Figure 3.3) and thus can make otherwise accepting superruns halt (E5). The disjoint nature of safe sets cannot be canceled without worsening complexity. Therefore we also change the rule on inclusions of the successors in



(a) The input SDTGBA



$\{q_n\}, \emptyset, \emptyset, \emptyset, \emptyset$
a
$\{q_n\}, \emptyset, \{q_1, f\}, \emptyset, \emptyset$
a
$\{q_n\}, \emptyset, \{q_1, q_2, f\}, \emptyset, \emptyset$
a
$\{q_n\}, \emptyset, \{q_1, f\}, \{q_2\}, \emptyset$
a
$\{q_n\}, \emptyset, \{q_1, q_2, f\}, \{q_2\}, \emptyset$
b
$\{q_n\}, \emptyset, \{q_1, f\}, \{f\}, \emptyset$
b

(b) A needed superrun halts for E5

Figure 3.3: Assuming $\mathcal{F} = \{F_0, F_1\}$, the consequence of the naive version of Optimization 3.4.2 is that the only potentially accepting superruns on $w = aaaab^\omega$ halt on intersecting runs in different safe sets (other superruns are non-accepting for E3 or E6). This is despite all runs of the input automaton being non-accepting.
The transition marked ⓪ in (a) belongs to F_0. In (b) the transitions marked 🟦 (would) belong to the only acceptance set of the complement automaton.

safe sets so that intersecting runs to appear in several safe sets $S_i, \dots, S_j$ must be kept in that one of them with the maximal index.

The ideas just presented are contained in the semantics of the following macro. Let $SAFE(i, A)$ be a macro defined

$$SAFE(i, A) = \big(\Delta(S_i, a, \delta \setminus F_i) \cup \Delta(S_{i-1}, a, F_{i-1}) \cup A\big) \setminus \bigcup_{j=i+1}^{k} S'_j$$

where $S_{-1} = F_{-1} = \emptyset$ (if $i = 0$). The set A is just kept as an option to extend the range of values within the intersection fix.

The optimization is formally defined by replacing E8, E9, E10 in Section 3.1 and adding a new one at the end:

E8) $S'_0 \subseteq SAFE\Big(0, \Delta(C, a, \bigcup \mathcal{F}) \cup \big(\Delta(N, a) \cap Q_D\big)\Big)$

(possible additions to S_0 are given by runs using an accepting transitions in C and runs entering Q_D),

E9) $S'_0 \supseteq SAFE(i, \emptyset)$

(runs in S_0 must stay there unless they use an accepting transition from F_0 or unless an intersection appears in some S_j for $j > 0$),

E10) $\Delta(S_k, a, F_k) = \emptyset$

(runs in S_k cannot use an accepting transition from F_k),

E11) $S'_i = SAFE(i, \emptyset)$ for each $1 \leq i \leq k$

(runs cannot escape from S_i except when destinations of accepting transitions from F_i are moved to S'_{i+1} if it exists; intersecting runs are kept in the safe set with the maximal index).

The reason why these changes preserve correctness is not easy to see at a first glance. After closer inspection, the optimization works because if a run visits each $F_0, \dots, F_k$ infinitely many times (it is accepting in the input automaton), it will always eventually be blocked in S_k or stay in C forever. If there is a set F_i that the run does not visit infinitely many times, there will be a proper time to move the run to S_0, from where it will get at worst[1] to S_i and no further. There it will never be blocked.

1. The run can in the mean time become safe relative to an F_j where $j < i$.

As a side note, the safe set with the maximal index has to be preferred for intersecting runs because otherwise it would be possible for an accepting run of the input automaton to avoid checking by jumping on intersecting runs to lesser sets just before being blocked in S_k.

3.4.3 Cutting Down S_i

To reduce the number of possible states appearing in a safe set S_i, there are at least two things we can do. The original NCSB serves as an inspiration because it forbids any accepting states in S. Since an accepting state can be translated as a source of only accepting transitions, the same optimization can be integrated into our extension.

The basic idea is straightforward: forbid sources of only accepting transitions from F_i in S_i, let us call these sources *accepting states in F_i*[2]. Since we work with generalized automata, this means that accepting states in each $F_i, \dots, F_j$ will not appear in any one of the corresponding safe sets $S_i, \dots, S_j$.

It is harder to define what exactly this means because we cannot just ignore states (E2 has to hold). The formal definition has to account for the following.

- If a new guess from C is an accepting state in F_i, then it can only appear in the safe set $S_{i'}$ with the minimal $i' \neq i$ such that it is not accepting in $S_{i'}$. Accepting states in *each* one of the declared acceptance sets $F_0, \dots, F_k$ cannot be moved to any safe set.
- When an accepting state in F_i is to appear in S_i, then it *must* instead appear in $S_{i'}$ as mentioned in the previous point except that $i' > i$ (otherwise runs could avoid checking by jumping on accepting states to lesser sets). If $S_{i'}$ does not exist, then such a superrun halts, especially when an accepting state in *each* one of the declared acceptance sets $F_0, \dots, F_k$ is to appear in S_i.

For the optimization already defined in Section 3.4.2, this will mean that a run can now jump not only on corresponding accepting transitions and intersections but also on accepting states, possibly through several successive safe sets at once. It is challenging to define

2. $F_i \neq \emptyset$ does *not* imply there exists an accepting state in F_i.

both optimizations so that they work hand in hand and not cancel each other out.

First we define the (possibly non-disjoint) sets $Acc_0, \dots, Acc_k$ for each declared acceptance set $F_0, \dots, F_k$ such that

$$Acc_i = \{q \in Q_D \mid \big((\{q\} \times \Sigma \times Q_D) \cap \delta\big) \setminus F_i = \emptyset\},$$

hence Acc_i contains accepting states in F_i. Then we define macros

- $M(i) = \{q \in \Delta(C, a, \bigcup F) \cup \big(\Delta(N, a) \cap Q_D\big)$
  $\mid \forall j < i : q \in Acc_j\}$,

  (set of candidates to be moved non-deterministically into S_i: such destinations of used accepting transitions or of used transit transitions that are accepting in each F_j where $j < i$),

- $K(i) = \{q \in Q_D \mid \exists l < i : q \in \Delta(S_l, a)$ and
  $\forall l \leq j < i : q \in Acc_j\}$,

  (set of candidates to be moved deterministically into S_i: successors to occur in S_l for some $l < i$ such that they are accepting in each S_j where $l \leq j < i$),

and then we alter the following rules from the optimization in Section 3.4.2.

E8) $S'_0 \subseteq SAFE\big(0, M(0)\big) \setminus Acc_0$

(possible additions to S_0 are such destinations of used accepting transitions or of used transit transitions that are not accepting in F_0 and they are not in an intersection with S_j where $j > 0$),

E9) $S'_i \supseteq SAFE\big(i, K(i)\big) \setminus Acc_i$ for each $0 \leq i \leq k$

(runs cannot escape S_i unless they use an accepting transition from F_i or an intersection appears in some S_j for $j > 0$; destinations of accepting transitions in S_i must be pushed to S'_{i+1} if it exists),

E10) $\Delta(S_k, a, F_k) = \emptyset \wedge \big(\Delta(S_k, a) \cap Acc_k = \emptyset\big) \wedge \big(K(k) \cap Acc_k\big) = \emptyset$

(safe runs in S_k cannot use a transition from F_k and cannot contain nor receive an accepting state in F_k), E11) $S'_i \subseteq SAFE\big(i, K(i) \cup M(i)\big) \setminus Acc_i$ for each $1 \leq i \leq k$

(possible additions to S_i are such destinations of accepting transitions or of transit transitions that are accepting in all F_j where $j < i$, unless they are also accepting in F_i or unless they are in an intersection with S_j where $j > i$).

From the formalism itself, it would be too verbose to prove that it preserves correctness. A sketch of the reasoning is this. Analogously to Section 3.4.2, an accepting run will stay forever in C or will eventually be blocked in S_k, possible jumping to greater safe sets on accepting states does not break this in any way.

On the other hand, for a non-accepting run containing finitely many transitions from F_i, there must be a proper time to move it to some S_j such that $j \leq i$ from where it will get at worst to S_i and no further because no remaining accepting transitions in F_i also imply no remaining successors accepting in F_i. There the run will never be blocked.

3.4.4 Optimizations Review

To sum up, let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \mathcal{F})$ be an SDTGBA where $Q = Q_N \cup Q_D$ such that the conditions for semi-determinism are met and $\mathcal{F} = \{F_0, \ldots, F_k\}$. For clarity, these are again the macros we have defined for the optimizations.

- $SAFE(i, A) = \big(\Delta(S_i, a, \delta \setminus F_i) \cup \Delta(S_{i-1}, a, F_{i-1}) \cup A\big) \setminus \bigcup_{j=i+1}^{k} S'_j$ where $S_{-1} = F_{-1} = \emptyset$ (if $i = 0$),
- $M(i) = \{q \in \Delta(C, a, \bigcup F) \cup \big(\Delta(N, a) \cap Q_D\big)$ $\mid \forall j < i : q \in Acc_j\}$,
- $K(i) = \{q \in Q_D \mid \exists l < i : q \in \Delta(S_l, a)$ and $\forall l \leq j < i : q \in Acc_j\}$.

The *optimized* complement automaton $\mathcal{C} = (R, \Sigma, \delta_{\mathcal{C}}, r_I, \{F_{\mathcal{C}}\})$ of $\mathcal{A}$ is defined as in Section 3.1, with the exception that the transition

$$(N, C, S_0, \ldots, S_k, B) \xrightarrow{a} (N', C', S'_0, \ldots, S'_k, B') \in \delta_{\mathcal{C}}$$

if and only if

E1) $N' = \Delta(N, a) \cap Q_N$,

E2) $C' \cup S'_0 \cup \ldots \cup S'_k$
$= \Delta(C \cup S_0 \cup \ldots \cup S_k, a) \cup \big(\Delta(N, a) \cap Q_D\big)$,

E3) $C' \supseteq \Delta(C, a, \delta \setminus \bigcup \mathcal{F})$,

E4) $C' \cap (S'_0 \cup \ldots \cup S'_k) = \emptyset$,

E5) $S'_i \cap S'_j = \emptyset$ for each $0 \leq i \neq j \leq k$,

E6) $\Delta(q, a) \neq \emptyset$ for each $q \in C$,

E7) if $\Delta(B, a) \cap C' \neq \emptyset$ then $B' = \Delta(B, a) \cap C'$ else $B' = C'$,

E8) $S'_0 \subseteq SAFE\big(0, M(0)\big) \setminus Acc_0$,

E9) $S'_i \supseteq SAFE\big(i, K(i)\big) \setminus Acc_i$ for each $0 \leq i \leq k$,

E10) $\Delta(S_k, a, F_k) = \emptyset \wedge \big(\Delta(S_k, a) \cap Acc_k = \emptyset\big) \wedge \big(K(k) \cap Acc_k\big) = \emptyset$,

E11) $S'_i \subseteq SAFE\big(i, K(i) \cup M(i)\big) \setminus Acc_i$ for each $1 \leq i \leq k$.

4 Evaluation

Although we have already analyzed the upper bound on the states of the output constructed by our extension in Section 3.2, experimental results can often help see more as to an algorithm's practical usefulness. What interested us mainly was the comparison with NCSB running on degeneralized automata because we wanted to see if our approach could save anything by skipping degeneralization. Then we were curious how our extension performs when compared with a completely another complementation algorithm able to work with the SDTGBA.

The SPOT library [6] met our requirements: for the testing data set it can generate random automata and also perform degeneralization we needed. It contains an implementation of NCSB able to work with the transition-based BA (TBA), hence transformation from TBA to BA was *not* needed. It also includes a tool that uses its own algorithm to complement any TGBA of choice based on *determinization*. Most importantly, SPOT C++ API provided us with an efficient platform to implement our extension. The overview of the CLI tools we used from SPOT can be found in Table 4.1.

Tool	Description
`randaut`	Generates random automata.
`randltl`	Generates random LTL formulas.
`ltl2tgba`	Converts an LTL formula into a TGBA.
`autfilt`	Filters automata based on given properties.
`autcross`	Checks and compares equivalent automata.

Table 4.1: Tools used in the evaluation.

4.1 Implementation (TGNCSB)

As said before, the extension's implementation is written in C++ using the SPOT API, and we call it TGNCSB (as in the *Transition-based Generalized* NCSB algorithm) to prevent confusion in the evaluation. It is attached and released under the GNU GPL license. Thanks to SPOT, its code is not long enough to need splitting into separate files. Therefore it consists of only two files: `tgncsb.cpp` and `tgncsb-opt.cpp` (excluding header files). The former represents the basic version formally defined in Section 3.1, and the latter is the optimized version reviewed in Section 3.4.4. Even though they have common parts, they are not merged in any way so that they can be explored with `diff` or a similar tool.

We must emphasize that TGNCSB is heavily based on the internal NCSB implementation present in SPOT[1] because editing a high-performance code should arguably be preferred to implementing from scratch. We have hopefully included sufficient commentary to make it readable and also made sure each rule from the formal construction is labeled in the corresponding part of the code.

A `CMakeLists.txt` file is included to simplify compilation. Therefore once SPOT is installed[2], `cmake . && make` should suffice to compile the implementation on a standard UNIX machine. Because making a stand-alone tool was not our aim, the resulting binaries simply take just one required and one optional argument (strictly in the given order): 1) the path to a file with the input automaton in the HOA format [7], and 2) `--postprocess` can be added if we want the output of TGNCSB to run through SPOT's optimizations before being returned (which is called *postprocessing*).

4.2 Experimental Comparison

Using SPOT's Python bindings, we have created a very simple Python script `degen_ncsb.py` which first runs degeneralization on a given input and only then it calls the internal NCSB (which is able to work directly with the TBA). This script also takes one required argument that is the path to a file with the input automaton in the HOA format, and again the optional second argument `--postprocess` can be added if we want postprocessing.

1. gitlab.lrde.epita.fr/spot/spot/blob/master/spot/twaalgos/complement.cc should point to the stable version.
2. See spot.lrde.epita.fr/install.html for instructions.

cid	command line
c1	`| autfilt --small`
c2	`| autfilt -v --acc-sets=0..1 | autfilt -v --is-deterministic`
c3	`| autfilt --unique -n 1000 --is-semi-deterministic`
c4	`| autcross --csv={path} -T 60 -t "{t1}cmd1" "{t2}cmd2"` ...

Table 4.2: Input processing.

The testing input can be divided into two types: the data set of automata generated by `randaut`, and the data set of automata generated using `ltl2tgba` on LTL formulas randomly generated by `randltl`. Both are explained in the corresponding sections. The testing input was then run through *preprocessing*.

The state spaces of input automata were reduced in all cases to prevent any algorithms from taking an advantage of input optimizations (c1). We wished to filter out trivially generalized automata ($|\mathcal{F}| = 1$) and deterministic automata (c2), since as already apparent in Section 3.2, both would obscure potential benefits of using our extension on strictly semi-deterministic non-trivially generalized automata. In the end, we enforced a selection of 1000 unique strictly semi-deterministic automata (c3).

The filtered and preprocessed input was then sent straight into `autcross` (c4). It takes a path where to save the results, the number of seconds to time out, and a series of strings, each specifying an arbitrary tool identifier (such as `t1`) used to label the corresponding result in the csv file and the command to be executed (such as `cmd1`). These were the specific tool strings used for calling `autcross`.

- `"{autfilt}autfilt --complement"`,
- `"{tgncsb}./tgncsb-opt %H > %O"`,
- `"{ncsb}./degen_ncsb.py %H > %O"`,
- `"{tgncsb}./tgncsb %H > %O"`.

The macro `%H` returns the temporary file path containing the tested automaton, the macro `%O` returns the path where `autcross` expects the output. Since `autfilt --complement` does postprocessing in its core, this algorithm was only included in the comparison when adding `--postprocess` after `%H`.

The main focus of measurings were output states and *edges*[3]. All experiments were running on a computer with the processor 1.6 GHz Intel Core i5 and 8 GB of RAM, while the timeout was set to 60 seconds. SPOT's version was 2.7.1.

4.2.1 Running On `randaut`

This data set was generated using `randaut 3..5 -n -1 -A 2..5 -Q8` (generating automata having $|Q| = 8$ and $2 \leq |\mathcal{F}| \leq 5$). It was piped straight into the tool chain presented in Table 4.2. We have placed the cumulative results in Table 4.3 for each one of the tools.

The scatter plots Figure 4.1, Figure 4.2 compare the number of generated states in particular instances (the x-th coordinate is used for the number of states generated for a given input automaton by one tool, the y-th for the other tool). The dotted areas are quite dense because the input data set consisted of automata reduced from the same number of states.

	states		edges	
tool / postprocessing	no	yes	no	yes
autfilt	-	17503	-	57843
tgncsb-opt	28624	23841	95026	78657
ncsb	36020	29443	103218	89388
tgncsb*	46117	-	203064	-

Table 4.3: Cells contain a cumulative number of states/edges in automata constructed for a test input of 1000 SDTGBA. The input was generated by `randaut`.
* Results for the unoptimized tgncsb are just illustrative here because they were not crosschecked due to their huge state spaces. Their postprocessing timed out in too many cases, hence tgncsb results with postprocessing are excluded.

3. Edges are merged transitions. For more information see https://spot.lrde.epita.fr/concepts.html#trans-edge.

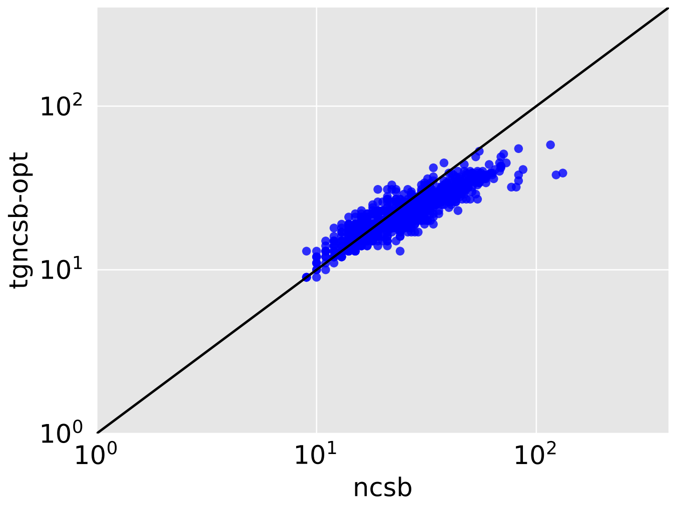


Figure 4.1: A comparison of the number of states created by tgncsb-opt and ncsb used on 1000 SDTGBA with postprocessing. Input automata were generated by `randaut`. Scales are logarithmic.

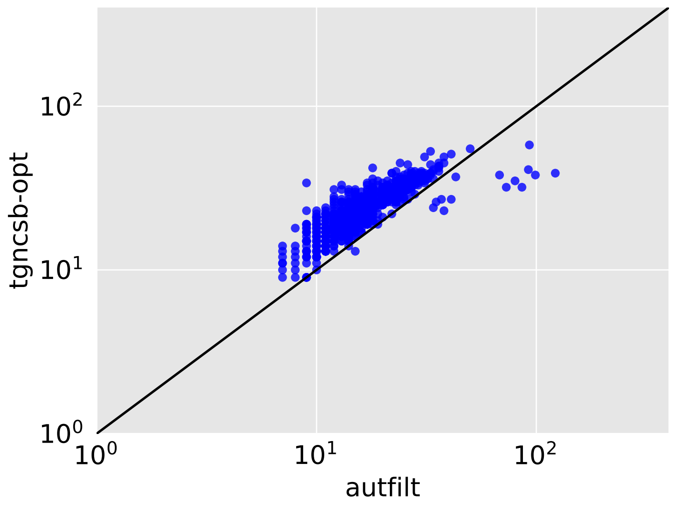


Figure 4.2: A comparison of the number of states created by tgncsb-opt and autfilt used on 1000 SDTGBA with postprocessing. Input automata were generated by `randaut`. Scales are logarithmic.

4.2.2 Running On `randltl`

This data set was created with `randltl -n -1 a b c d | ltl2tgba` (generating random LTL formulas on four atomic propositions, then converting the formulas into the TGBA). It was piped straight into the toolchain presented in Table 4.2.

The toolchain makes sure that only those LTL formulas are used which are converted into a valid and strictly semi-deterministic TGBA with $|\mathcal{F}| \geq 2$. It also makes sure that 1000 unique instances are always received (even if some formulas fail to translate to the required form).

Table 4.3 contains the cumulative results measured for each one of the tools. To repeat ourselves, since `autfilt --complement` uses post-processing in its core, it was not included in the measurings without `--postprocess`.

For a better insight check Figure 4.3 and Figure 4.4. These scatter plots again compare the number of generated states in particular instances (hence the x-th coordinate is used for the number of states generated for a given input automaton by one tool, the y-th for the other tool). The dotted areas are less dense here than their counterparts in Section 4.2.1 because LTL to TGBA translation gave a more varying number of states in the input automata.

	states		edges	
tool / postprocessing	no	yes	no	yes
autfilt	-	10534	-	49416
tgncsb-opt	33943	12289	200278	49564
ncsb	41090	12657	207245	49262
tgncsb	105509	14454	824408	62004

Table 4.4: Cells contain a cumulative number of states/edges in automata constructed for a test input of 1000 SDTGBA. The input was generated by converting `randltl` formulas to the required SDTGBA.

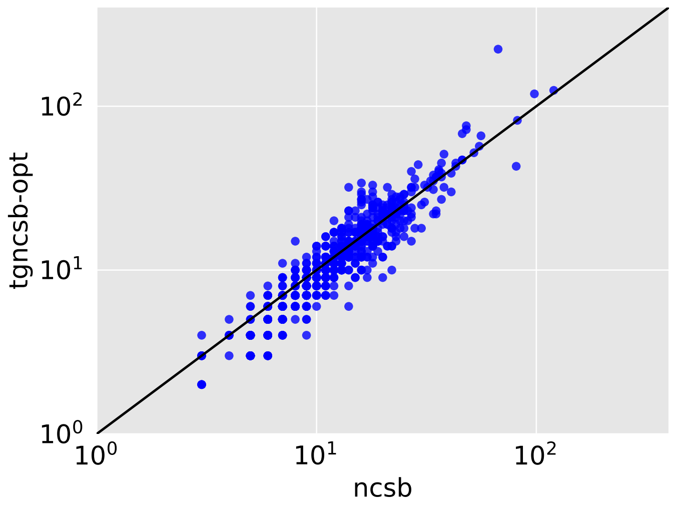


Figure 4.3: A comparison of the number of states created by tgncsb-opt and ncsb with postprocessing. Input automata were generated by converting random LTL formulas to the SDTGBA. Scales are logarithmic.

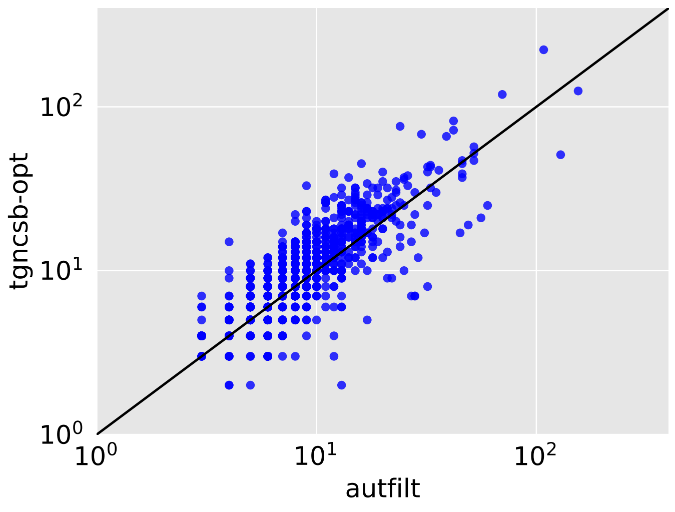


Figure 4.4: A comparison of the number of states created by tgncsb-opt and autfilt with postprocessing. Input automata were generated by converting random LTL formulas to the SDTGBA. Scales are logarithmic.

4.2.3 Observations

The optimized TGNCSB (tgncsb-opt) clearly outperforms NCSB run on degeneralized automata both with and without postprocessing. Its advantage in automata generated from random LTL formulas is smaller probably because the automata from this input turned out to have at most $|\mathcal{F}| = 3$, and that only 28 out of 1000 times. Therefore degeneralization was less costly, and thus not much could have been saved in contrast with input automata which had a non-trivial number of acceptance sets.

However, it seems SPOT's `autfilt --complement` still has the edge over TGNCSB in most cases despite its generality, which remains an open question for further analysis. Another interesting thing to observe is the vertical pattern present in Figure 4.3 and Figure 4.4. In some groups of LTL formulas, it gets worse only for TGNCSB and not for NCSB or `autfilt --complement`. One hypothesis might be that there are specific classes of LTL formulas that TGNCSB cannot handle very well, which could also suggest a direction for further improvement of TGNCSB.

5 Conclusion

Building upon the NCSB algorithm, we have devised its extension for complementation of the SDTGBA. The extension has been formally defined, its complexity analyzed, and its correctness has been proven. Afterwards, several key optimizations have been described and defined to improve its per instance performance.

The extension can complement the SDTGBA without transforming the input automaton, and also produces trivially generalized automata that work the same as the conventional (transition-based) BA. We have also created an implementation of the extension in SPOT, which was called TGNCSB. It was used to evaluate experimentally how much the extension can save compared with two standard complementation algorithms for the SDTGBA: the original NCSB using degeneralization and the main SPOT's complementation algorithm based on determinization called `autfilt --complement`. We have concluded that the optimized TGNCSB outperforms NCSB running on degeneralized automata.

The extension does not perform so well when compared with `autfilt --complement`. However, even though these results do not look so promising, they suggest there might still be room for improvement. A vertical pattern appeared in the scatter plot comparison of TGNCSB and the other two algorithms used on automata generated by converting random LTL formulas to the SDTGBA.

This means some groups of LTL formulas are worse for the optimized TGNCSB than others. Further analysis has to be made as to why these groups appear and whether the formulas they contain have anything in common. It also needs to be found out why `autfilt --complement` has such an edge over TGNCSB despite it being a very generic algorithm.

Bibliography

1. TSAI, Ming-Hsien; FOGARTY, Seth; VARDI, Moshe Y.; TSAY, Yih-Kuen. State of Büchi Complementation. In: DOMARATZKI, Michael; SALOMAA, Kai (eds.). *Implementation and Application of Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 264. ISBN 978-3-642-18098-9. Available from DOI: `10.1007/978-3-642-18098-9_28`.

2. BLAHOUDEK, František. *Automata for Formal Methods: Little Steps Towards Perfection* [online]. 2018 [visited on 2019-04-21]. Available from: `https://theses.cz/id/8z875b/`. PhD thesis. Masarykova univerzita, Fakulta informatiky, Brno.

3. BLAHOUDEK, František; HEIZMANN, Matthias; SCHEWE, Sven; STREJČEK, Jan; TSAI, Ming-Hsien. Complementing Semi-deterministic Büchi Automata. In: CHECHIK, Marsha; RASKIN, Jean-François (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 770–787. ISBN 978-3-662-49674-9. Available from DOI: `10.1007/978-3-662-49674-9_49`.

4. CHEN, Yu-Fang; HEIZMANN, Matthias; LENGÁL, Ondřej; LI, Yong; TSAI, Ming-Hsien; TURRINI, Andrea; ZHANG, Lijun. Advanced Automata-based Algorithms for Program Termination Checking. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, PA, USA: ACM, 2018, pp. 143–144. PLDI 2018. ISBN 978-1-4503-5698-5. Available from DOI: `10.1145/3192366.3192405`.

5. BLAHOUDEK, František; DURET-LUTZ, Alexandre; KLOKOČKA, Mikuláš; KŘETÍNSKÝ, Mojmír; STREJČEK, Jan. Seminator: A Tool for Semi-Determinization of Omega-Automata. In: EITER, Thomas; SANDS, David (eds.). *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, 2017, vol. 46, pp. 356–367. EPiC Series in Computing. ISSN 2398-7340. Available from DOI: `10.29007/k5nl`.

6. DURET-LUTZ, Alexandre; LEWKOWICZ, Alexandre; FAUCHILLE, Amaury; MICHAUD, Thibaud; RENAULT, Etienne; XU, Laurent. Spot 2.0 — a framework for LTL and ω-automata manipulation.

In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Springer, 2016, vol. 9938, pp. 122–129. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-319-46520-3_8`.

7. BABIAK, Tomáš; BLAHOUDEK, František; DURET-LUTZ, Alexandre; KLEIN, Joachim; KŘETÍNSKÝ, Jan; MÜLLER, David; PARKER, David; STREJČEK, Jan. The Hanoi Omega-Automata Format. In: KROENING, Daniel; PĂSĂREANU, Corina S. (eds.). *Computer Aided Verification*. Cham: Springer International Publishing, 2015, pp. 479–486. ISBN 978-3-319-21690-4.

A Attachments

Attached are the following files

- `tgncsb.zip` - an archive containing the latest version of TGNCSB and its optimized version,
- `experiments.zip` - an archive containing the NCSB script working with the SDTGBA and all the measured results including their description.