Learning Büchi Automata and Its Applications

Chapter · April 2019

DOI: 10.1007/978-3-030-17601-3_2

Yong Li
Chinese Academy of Sciences
19 PUBLICATIONS 51 CITATIONS
SEE PROFILE

Andrea Turrini
Chinese Academy of Sciences
41 PUBLICATIONS 291 CITATIONS
SEE PROFILE

Yu-Fang Chen
Academia Sinica
59 PUBLICATIONS 697 CITATIONS
SEE PROFILE

Lijun Zhang
Chinese Academy of Sciences
99 PUBLICATIONS 1,567 CITATIONS
SEE PROFILE

Learning Büchi Automata and Its Applications

Yong Li[1,2] , Andrea Turrini[1,3] , Yu-Fang Chen[4] , Lijun Zhang[1,2]

[1] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences, Beijing
[2] University of Chinese Academy of Sciences, Beijing
[3] Institute of Intelligent Software, Guangzhou
[4] Institute of Information Science, Academia Sinica, Taipei

Abstract. In this work, we review an algorithm that learns a Büchi automaton from a teacher who knows an ω-regular language; the algorithm is based on learning a formalism named *family of DFAs* (FDFAs) recently proposed by Angluin and Fisman. We introduce the learning algorithm by learning the simple ω-regular language $(ab)^\omega$: besides giving the readers an overview of the algorithm, it guides them on how the algorithm works step by step. Further, we demonstrate how the learning algorithm can be exploited in classical automata operations such as complementation checking and in the context of termination analysis.

1 Introduction

Model checking is a widely used technique in the verification of hardware and software systems, scaling from case studies in academic publications to real systems in industry; the importance of model checking has been recognized by means of the 2007 Turing award, which has been assigned to Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis for "their roles in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries".

Large systems are usually obtained by developing several small components that interact concurrently with each other so to globally achieve the desired functionality. The main obstacle in applying model checking to concurrent systems is the well-known state explosion problem [31]. The number of global states of such systems can be enormous: it is actually of the form n^p where p is the number of processes and n is the number of states in each process. There have been several approaches proposed in literature to combat the state explosion problem, such as symbolic model checking based on BDDs [78], bounded model checking [18], and learning-based compositional verification [34]. The latter approach, the learning-based compositional verification, tries to learn models of the single components that are smaller than the original processes while preserving their behavior. The learning algorithm used in [34] is the well-known L^* algorithm proposed by Dana Angluin [8], which allows one to learn deterministic finite automata (DFAs).

Automata learning algorithms have received significant attention from the verification community in the past two decades. Besides being used to improve the efficiency and scalability of compositional verification, automata learning has also been successfully applied in other aspects of verification: among others, it has been used to automatically generate interface models of computer programs [7], to learn a model of the traces of the system errors for diagnosis purposes [27], to find bugs in the implementation of network protocols [39], to extract behavior model of programs for statistical program analysis [29], and to do model-based testing and verification [81, 107]. Later in 2017, Frits Vaandrager [102] explains the concept of *model learning* used in the above applications.

In order to be of practical use, the learning algorithms have to be computationally efficient and easily adaptable to the different learning scenarios. On the one hand, with more complex tasks at hand, some researchers have proposed several optimizations to improve the efficiency of finite automata learning, such as learning algorithms based on *classification trees* [59,63], efficient counterexample analysis for learning algorithms [87], learning algorithm NL^* for nondeterministic finite automata (NFAs) [20], and learning algorithms for alternating finite automata [10].

On the other hand, due to the demands from the different verification tasks, some researchers also develop and apply learning algorithms for richer models. For example, there are learning algorithms for I/O automata [2], event-recording automata [51], register automata [57, 58], timed systems [75], probabilistic systems [44], and nominal automata [79]. Specially, van Heerdt *et al.* in [103] proposed an automata learning framework based on category theory which unifies the learning of several automata including DFAs and weighted automata.

However, aforementioned learning algorithms are all designed for the automata accepting finite words; those automata are used to model the finite behaviors of the systems, which are usually characterized by safety properties expected to hold. For instance, one can use a DFA to recognize all possible bad behaviors, i.e., behaviors leading in a finite number of steps to a state violating a safety property. Instead, for characterizing the infinite behaviors of the systems, generally corresponding to liveness properties, automata accepting infinite words are used.

In his seminal work [25], Julius Richard Büchi introduced automata accepting infinite words to prove the decidability of a restricted *monadic second order* (MSO) logic; now such automata are widely known as Büchi automata (BA). A Büchi automaton has the same structure as an NFA, except that it operates on infinite words: instead of accepting a finite word if it leads the run of the automaton to end in an accepting state, a BA accepts an infinite word if it leads the automaton to visit an accepting state infinitely often. Büchi automata are nowadays very popular in the model checking field, in particular when the specification is given by a linear temporal logic (LTL) formula; see the introductory paper by Vardi [104] on the use of BAs for LTL analysis.

Besides being used in LTL verification and synthesis, Büchi automata have been also used as a standard model to describe the liveness properties of distributed systems [6]. Therefore, in order to verify whether a concurrent system satisfies a liveness property, one can model every process of the system as a Büchi automaton. It follows that if one can learn smaller Büchi automata for the processes, then performing compositional verification on the given concurrent system can become less expensive, similarly to the DFA case. Motivated by that, Farzan *et al.* presented in [43] the first learning algorithm for the complete-class of ω-regular languages represented as Büchi automata; the algorithm is able to extract automatically a Büchi automaton as an assumption from a component of concurrent systems for compositional verification. Note that already in 1993 Maler and Pnueli [77] introduced the first learning algorithm for Büchi automata, but it learns Büchi automata accepting only a proper subset of ω-regular languages. In 2014, Angluin and Fisman proposed in [11] a learning algorithm for the ω-regular languages by means of a formalism called a *family of DFAs* (FDFAs). Later in [73], Li *et al.* proposed to use classification trees to learn FDFAs rather than observation tables used by Angluin and Fisman. Learning algorithms based on classification trees usually need less runtime memory and can be much more efficient when compared to its observation table based counterparts [59]. Further, Li *et al.* presented in [73] a more efficient learning algorithm for Büchi automata based on FDFAs and classification trees compared to the learning algorithm in [43].

There are already a few learning algorithms for Büchi automata available in the literature, yet the learning algorithms are not widely used in the model checking community. One reason for this is that the learning algorithms are quite technically demanding and not so easy to follow and understand; in this paper we give a simple presentation of one BA learning algorithm, with simple but complete examples, to introduce the reader to such learning framework. Another reason is that there are fewer learning libraries available for Büchi automata compared to those implemented for learning automata accepting finite words: for instance, for learning automata accepting finite words there are robust and publicly available libraries such as libalf [21] and LearnLib [60]. To the best of our knowledge, there is, however, only one publicly available library for learning Büchi automata named ROLL [73] which implements also the BA learning algorithm described in this paper.

In this paper, we review the BA learning algorithm proposed in [73] by learning the simple ω-regular language $(ab)^\omega$: besides giving the reader an overview of the algorithm, it guides them on how the algorithm works step by step. Our main goal in this work is to give an intuitive explanation of the different learning algorithms for both finite and ω-regular languages; in this way the reader can get the ideas underlying the learning algorithms before getting involved in their formalism, presented in the related literature; we achieve this by means of the examples we carefully chose so to be simple but still exposing the different challenges the learning algorithms for Büchi automata face and the solution techniques that have been adopted.

Further, we discuss two possible interesting applications of the BA learning algorithms. The complementation problem for Büchi automata is a challenging problem in the research community in theory and practice. We show that the BA learning algorithm can be easily applied to complement Büchi automata. Experimental results show that the learning-based complementation algorithm of Büchi automata can yield much smaller complement automata for some cases than classical algorithms. Lastly, we discuss how the learning algorithms can be also applied in proving the termination of C programs. Heizmann *et al.* in [55] proposed a novel termination analysis algorithm based on Büchi automata. Interestingly, the efficiency and scalability of this termination analysis algorithm highly depend on getting smaller complement automata of Büchi automata, where one naturally can use the learning based complementation algorithm.

Organization of the paper. We first set up some notions and notations for this work in Section 2. We then introduce some basic operations on Büchi automata in Section 3, together with their complexity analysis, before turning to the learning algorithms in the following sections. In order to ease the presentation, we first present the learning algorithm for DFAs in Section 4 and then move onto the learning of Büchi automata in Section 5. After that, we show how to apply our learning algorithm to the complementation problem of Büchi automata in Section 6. Before concluding the paper in Section 8, we consider the application of BA learning algorithm to program termination analysis in Section 7.

2 Preliminaries

Let X and Y be two sets; we use $X \ominus Y$ to denote their *symmetric difference*, i.e., the set $(X \setminus Y) \cup (Y \setminus X)$. We use $[i \cdots j]$ to denote the set $\{i, i+1, \ldots, j\}$.

Let Σ denote a finite non-empty set of letters called *alphabet*. A *word* is a finite or infinite sequence $w = w_1 w_2 \cdots$ of letters in Σ; we denote by $|w|$ the length of the word w, i.e., the number letters in w. If w is infinite, then $|w| = \infty$, and we call it an *ω-word*. We use ε to denote the word of length 0, i.e., the empty word. We denote by Σ^* and Σ^ω the sets of all finite and infinite words, respectively. Moreover, we use Σ^+ to represent the set $\Sigma^* \setminus \{\varepsilon\}$.

We denote by $w[i]$ the i-th letter of a word w. We use $w[i..k]$ to denote the subword of w starting at the i-th letter and ending at the k-th letter, inclusive, when $i \leq k$ and the empty word ε when $i > k$. For $u \in \Sigma^*$, we denote by $\mathrm{Pref}(u)$ the set of its prefixes, i.e., $\mathrm{Pref}(u) = \{\varepsilon, u[1], u[1..2], \ldots, u[1..|u|]\}$. Similarly, we denote by $\mathrm{Suf}(u)$ the set of its suffixes, i.e., $\mathrm{Suf}(u) = \{u[1..|u|], u[2..|u|], \ldots, u[|u|], \varepsilon\}$. Given a finite word $u = u_1 \cdots u_k$ and a word w, we denote by $u \cdot w$ the *concatenation* of u and w, i.e., the finite or infinite word $u \cdot w = u_1 \cdots u_k w_1 \cdots$. We may just write uw instead of $u \cdot w$.

Definition 1. *An* acceptor automaton *is a tuple $A = (\Sigma, Q, \bar{q}, \delta, F)$ consisting of the following components: a finite alphabet Σ, a finite set Q of states, an initial state $\bar{q} \in Q$, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, and an accepting condition F.*

For convenience, we also use $\delta(q, a)$ to denote the set $\{\, q' \in Q \mid (q, a, q') \in \delta \,\}$.

In the remainder of the paper, we assume that all automata share the same alphabet Σ, which we may omit from their definitions.

A *run* of an acceptor automaton on a finite word $v = a_1 a_2 a_3 \cdots a_n$, $n \geq 1$, is a sequence of states $q_0, q_1, \dots, q_n$ such that $q_0 = \bar{q}$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for every $0 \leq i < n$; similarly, a *run* of an acceptor automaton on an infinite word $w = a_1 a_2 a_3 \cdots$ is a sequence of states $q_0, q_1, \dots$ such that $q_0 = \bar{q}$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for each $i \in \mathbb{N}$. The run on a word is *accepting* if it satisfies the accepting condition F. A word is accepted by an acceptor automaton A if A has an accepting run on it.

A *finite language* is a subset of Σ^* while an *ω-language* is a subset of Σ^ω; the language of an acceptor automaton A, denoted by $\mathcal{L}(A)$, is the set $\{\, u \in \Sigma^* \cup \Sigma^\omega \mid u \text{ is accepted by } A \,\}$.

A *deterministic acceptor automaton* is an acceptor automaton such that $|\delta(q, a)| \leq 1$ for any $q \in Q$ and $a \in \Sigma$. For deterministic acceptor automata, we may write $\delta(q, a) = q'$ instead of $\delta(q, a) = \{q'\}$. The transition relation of a deterministic acceptor automaton can be lifted to finite words by defining $\delta(q, \varepsilon) = q$ and $\delta(q, av) = \delta(\delta(q, a), v)$ for each $q \in Q$, $a \in \Sigma$, and $v \in \Sigma^*$. We also use $A(v)$ as a shorthand for $\delta(\bar{q}, v)$.

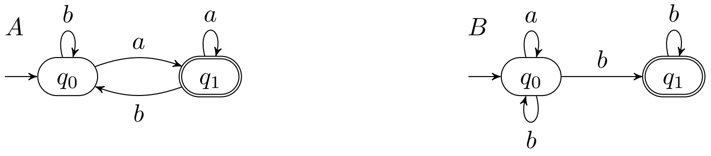
A *finite automaton* (FA) is an acceptor automaton where $F \subseteq Q$ and a finite word v is accepted if there is a run $q_0, q_1, \dots, q_n$ on v such that $q_n \in F$; no infinite word is accepted. A *deterministic finite automaton* (DFA) is a FA which is also a deterministic acceptor automaton. A complement DFA $A^{\mathcal{C}}$ of a DFA A is a DFA such that $\mathcal{L}(A^{\mathcal{C}}) = \Sigma^* \setminus \mathcal{L}(A)$. Complementing a DFA is easy: it is enough to add an accepting sink state collecting all missing transitions and complement the original set of accepting states. Let A and B be two FAs; one can construct a product FA, denoted by $A \times B$, accepting the language $\mathcal{L}(A) \cap \mathcal{L}(B)$ using a standard product construction; see, e.g., [56].

A *Büchi automaton* (BA) is an acceptor automaton where $F \subseteq Q$ and an infinite word w is accepted if there is a run $\rho = q_0, q_1, \dots$ on w such that for each $i \in \mathbb{N}$, there exists $j > i$ such that $\rho[j] \in F$; no finite word is accepted. Intuitively, an infinite word w is accepted by a BA if there exists a run on w visiting at least one accepting state in F infinitely often. A *deterministic Büchi automaton* (DBA) is a BA which is also a deterministic acceptor automaton.

A BA is a *limit deterministic Büchi automaton* (LDBA) if its set of states Q can be partitioned into two disjoint sets Q_N and Q_D, such that 1) $\delta(q, a) \subseteq Q_D$ and $|\delta(q, a)| \leq 1$ for each $q \in Q_D$ and $a \in \Sigma$, and 2) $F \subseteq Q_D$. It is trivial to note that each DBA is also an LDBA, by taking $Q_N = \emptyset$ and $Q_D = Q$.

Example 1. As examples of Büchi automata, consider the two automata shown in Figure 1. The automaton A is a DBA with alphabet $\Sigma = \{a, b\}$, set of states $Q = \{q_0, q_1\}$, initial state $\bar{q} = q_0$ (marked by the small incoming arrow), transition relation $\delta = \{(q_0, a, q_1), (q_0, b, q_0), (q_1, a, q_1), (q_1, b, q_0)\}$, and $F = \{q_1\}$ (denoted as a double-circled state). The language accepted by A is the ω-regular language $\mathcal{L}(A) = \{\, w \in \Sigma^\omega \mid w \text{ has infinitely many } a\text{'s} \,\}$.

The automaton B is an NBA that accepts the language $\mathcal{L}(B) = \{\, w \in \Sigma^\omega \mid w \text{ has finitely many } a\text{'s} \,\}$, which is the complement of $\mathcal{L}(A)$. Note that B is also



$\mathcal{L}(A) = \{\, w \mid w \text{ has infinitely many } a\text{'s} \,\}$ $\mathcal{L}(B) = \{\, w \mid w \text{ has finitely many } a\text{'s} \,\}$

Fig. 1. Examples of Büchi automata and their accepted languages

a limit deterministic Büchi automaton, where the corresponding partition of Q is given by $Q_N = \{q_0\}$ and $Q_D = \{q_1\}$.

We call the language of an FA a *regular* language. An ω-language $L \subseteq \Sigma^\omega$ is *ω-regular* if there exists a BA A such that $L = \mathcal{L}(A)$. Words of the form uv^ω, where $u \in \Sigma^*$ and $v \in \Sigma^+$, are called *ultimately periodic* words. We use a pair of finite words (u, v) to denote the ultimately periodic word $w = uv^\omega$. We also call (u, v) a *decomposition* of w; note that an ultimately periodic word can have several decompositions: for instance (u, v), (uv, v), and (u, vv) are all decompositions of uv^ω. For an ω-language L, let $\mathrm{UP}(L) = \{\, uv^\omega \in L \mid u \in \Sigma^*, v \in \Sigma^+ \,\}$ denote the set of all ultimately periodic words in L. Note that the set of ultimately periodic words of an ω-regular language L can be seen as the fingerprint of L, as stated by the following theorem.

Theorem 1 (Ultimately Periodic Words of ω-Regular Languages [25, 26]). *(1) Every non-empty ω-regular language L contains at least one ultimately periodic word. (2) Let L, L' be two ω-regular languages. Then $L = L'$ if and only if $UP(L) = UP(L')$.*

We refer interested reader to [25,26] for the proof of Theorem 1. An immediate consequence of Theorem 1 is that, for any two ω-regular languages L and L', if $L \neq L'$ then there must exist some ultimately periodic word $uv^\omega \in \mathrm{UP}(L) \ominus \mathrm{UP}(L')$.

3 Operations on Büchi Automata

In this section we present how nondeterministic Büchi automata support the standard set operations on their languages, namely, union, intersection, and complementation, as well as derived operations and decision problems. The main result is that nondeterministic Büchi automata are closed under such operations, e.g., giving two Büchi automata A_0 and A_1, we can construct another Büchi automaton A such that $\mathcal{L}(A) = \mathcal{L}(A_0) \cap \mathcal{L}(A_1)$. Deterministic Büchi automata, however, are strictly less expressive than nondeterministic ones, since there are ω-regular languages accepted by a nondeterministic BA for which there does not exist a deterministic BA accepting them; DBAs are also *not* closed under complementation, i.e., there is a DBA whose complement language can only be accepted by a nondeterministic BA.

There are several resources available in literature for the readers interested in more details on ω-languages and their automata; see, e.g., [30,32,50,62,93,95,96].

3.1 Union of Büchi Automata

Given two Büchi automata A_0 and A_1, it is rather easy to construct a Büchi automaton $A_{0\cup1}$ such that $\mathcal{L}(A_{0\cup1}) = \mathcal{L}(A_0) \cup \mathcal{L}(A_1)$. In fact, since by definition of language of a Büchi automaton, a word w belongs to its language if there exists an accepting run on w, it is enough to create an automaton having all runs of A_0 and A_1: this can be easily achieved by just considering A_0 and A_1 as a single automaton, up to some minor adaptation on the initial state.

Proposition 1. *Given two Büchi automata $A_0 = (Q_0, \bar{q}_0, \delta_0, F_0)$ and $A_1 = (Q_1, \bar{q}_1, \delta_1, F_1)$ such that $Q_0 \cap Q_1 = \emptyset$, let $A_{0\cup1} = (Q, \bar{q}, \delta, F)$ be the Büchi automaton whose components are defined as follows:*

- *$Q = Q_0 \cup Q_1 \cup \{\bar{q}\}$ where $\bar{q}$ is a fresh state such that $\bar{q} \notin Q_0 \cup Q_1$,*
- *$\delta = \delta_0 \cup \delta_1 \cup \{\, (\bar{q}, a, q_0) \mid q_0 \in \delta_0(\bar{q}_0, a) \,\} \cup \{\, (\bar{q}, a, q_1) \mid q_1 \in \delta_1(\bar{q}_1, a) \,\}$, and*
- *$F = F_0 \cup F_1$.*

Then, $\mathcal{L}(A_{0\cup1}) = \mathcal{L}(A_0) \cup \mathcal{L}(A_1)$ with $|Q| = |Q_0| + |Q_1| + 1$.

The proof of the above proposition is rather trivial: given an ω-word w, except for the initial state $\bar{q}$, a run on w of the automaton $A_{0\cup1}$ is identical to a run on w of either A_0 or A_1.

Note that the requirement that A_0 and A_1 must have disjoint sets of states can be easily fulfilled by simply renaming their states, since actual state names play no role in accepting a word. Moreover, if we would have allowed a set of initial states instead of a single initial state, then the union automaton would be just the component-wise union of the two given Büchi automata.

3.2 Intersection of Büchi Automata

The construction of a Büchi automaton accepting the intersection of the languages of A_0 and A_1 is slightly more involved than their union. The main idea underlying the intersection construction is to run on the input word in parallel in both A_0 and A_1, by means of a product construction similar to the one for the intersection of finite automata; as accepting condition, we require that we reach the accepting states of A_0 and A_1 in an alternating mode, i.e., every time we reach an accepting state in A_c for $c \in \{0, 1\}$, then we have to reach an accepting state in A_{1-c}. If we can alternate infinitely often, then both automata accept the input word, i.e., it is in the intersection of their languages; if we alternate only finitely often, this means that the BA where we get stuck is not accepting such a word, so the intersection automaton must reject the word as well. This is different from the accepting condition for finite automata, where a product state is accepting if both states in the pair are accepting: in fact, for infinite words it does not matter whether the two BAs reach an accepting state exactly at the same moment, since it can also be the case that both automata accept an ω-word w but A_0 reaches an accepting state only once every ten times A_1 has reached an accepting state.

Proposition 2. *Given two Büchi automata* $A_0 = (Q_0, \bar{q}_0, \delta_0, F_0)$ *and* $A_1 = (Q_1, \bar{q}_1, \delta_1, F_1)$*, let* $A_{0\cap 1} = (Q, \bar{q}, F, \delta)$ *be the Büchi automaton whose components are defined as follows:*

- $Q = Q_0 \times Q_1 \times \{0, 1\}$*;*
- $\bar{q} = (\bar{q}_0, \bar{q}_1, 0)$*;*
- $\delta = \{\, ((q_0, q_1, c), a, (q'_0, q'_1, \mathit{next}(q_0, q_1, c))) \mid q'_0 \in \delta_0(q_0, a), q'_1 \in \delta_1(q_1, a) \,\}$ *where* $\mathit{next}\colon Q_0 \times Q_1 \times \{0, 1\} \to \{0, 1\}$ *is defined as*

$$\mathit{next}(q_0, q_1, c) = \begin{cases} 1 - c & \textit{if } q_c \in F_c, \\ c & \textit{otherwise;} \end{cases}$$

- $F = F_0 \times Q_1 \times \{0\}$.

Then, $\mathcal{L}(A_{0\cap 1}) = \mathcal{L}(A_0) \cap \mathcal{L}(A_1)$ *with* $|Q| = 2 \cdot |Q_0| \cdot |Q_1|$.

The above construction is based on the transformation of generalized Büchi automata to Büchi automata. Generalized BAs differ from BAs only on the fact that they have multiple accepting sets; an ω-word w is accepted if there exists a run on w reaching a state in each accepting set infinitely often. Since generalized BAs have the same expressive power as ordinary BAs and are not used in this work, we refer the interested reader to, e.g., [32] for more details.

3.3 Complementation of Büchi Automata

Complementing Büchi automata is the most difficult operation on their languages. First of all, the usual subset construction used for converting nondeterministic finite automata to equivalent deterministic finite automata and then easily complement the resulting DFAs can not be adapted to Büchi automata since DBAs are strictly less expressive than BAs:

Proposition 3 (cf. [68]). *There exists an* ω*-regular language* L *that is recognizable by a BA but not by a DBA.*

This means that for such a language L, we can find a BA A such that $\mathcal{L}(A) = L$ but there does not exist a DBA D such that $\mathcal{L}(D) = L$. As a consequence, applying a subset construction to A does not lead to a DBA accepting the same language.

Note that the language witnessing the correctness of the above result is rather simple: $L = \Sigma^* \cdot b^\omega$, that is, L is the language of all words having only b occurring infinitely often. For $\Sigma = \{a, b\}$, this language is recognized by the BA B shown in Figure 1; its complement, i.e., the language whose words contain infinitely many

a, is easily recognized by the DBA A also shown in Figure 1. This means that DBAs are *not* closed under complementation, while BAs are indeed closed, as witnessed by the several complementation algorithms that have been proposed in literature.

Before presenting such algorithms, we want to introduce the main result about the complexity of complementing Büchi automata.

Proposition 4 (cf. [89]). *Given a BA A with n states, it is possible to construct a BA $A^{\mathcal{C}}$ such that $\mathcal{L}(A^{\mathcal{C}}) = \Sigma^{\omega} \setminus \mathcal{L}(A)$ whose number of states is in $\Omega(tight(n-1))$ and $\mathcal{O}(tight(n+1))$, where $tight(n) \approx (0.76n)^n$.*

In practice, the above is the best known complexity result for the complementation of Büchi automata, where the lower- and upper-bounds about the number of states of the complement Büchi automaton have a minor gap lying in $\mathcal{O}(n^2)$.

There are mainly four types of complementation algorithms, according to the classification proposed in [19,97]: Ramsey-based [24,25,92], rank-based [47, 52,65,89], determinization-based [45,82,88,90], and slice-based [5,61,97,106] complementation. A complementation construction unifying the rank-based and slice-based approaches can be found in [46]. All these algorithms construct the complement Büchi automata based on the transition structures of the input Büchi automata. Besides the complementation algorithm proposed for nondeterministic Büchi automata, there are also complementation algorithms specialized for limit deterministic Büchi automata [19,28] and for deterministic Büchi automata [66].

Given the highly demanding technicalities involved in the above complementation algorithms for Büchi automata, we refer the interested reader to the cited literature for more details on the different approaches and algorithms.

3.4 Difference of Büchi Automata

The BA language difference operation is tightly connected to the complementation operation, from which it derives its super-exponential complexity, as stated by the following proposition.

Proposition 5. *Given two BAs A_0 and A_1 with n_0 and n_1 states, respectively, it is possible to construct a BA $A_{0\setminus 1}$ such that $\mathcal{L}(A_{0\setminus 1}) = \mathcal{L}(A_0) \setminus \mathcal{L}(A_1)$ whose number of states is in $\Omega(n_0 \cdot tight(n_1 - 1))$ and $\mathcal{O}(n_0 \cdot tight(n_1 + 1))$.*

The language difference operation is based on the complementation operation: in order to get an automaton $A_{0\setminus 1}$ such that $\mathcal{L}(A_{0\setminus 1}) = \mathcal{L}(A_0) \setminus \mathcal{L}(A_1)$, it is enough to construct the automaton for the language $\mathcal{L}(A_0) \cap \mathcal{L}(A_1^{\mathcal{C}})$. Thus, the complexity result follows from Propositions 2 and 4.

Note that we can not improve the complexity of the language difference operation to be better than $\Omega(tight(n_1 - 1))$, since otherwise we would be able to improve the complexity of the complementation operation as well, since trivially we have that $\mathcal{L}(A_1^{\mathcal{C}}) = \Sigma^{\omega} \setminus \mathcal{L}(A_1)$ where Σ^{ω} is the language of the BA A_0 having exactly one state, the initial state, being accepting with only self-loops as transitions, so in Proposition 8 we would have $n_0 = 1$.

3.5 Decision Problems on Büchi Automata

Besides the three main operations presented above, namely union, intersection, and complementation, there are three main decision problems relative to the languages of Büchi automata: emptiness, universality, and language inclusion.

Given a BA A, the emptiness problem is relative to decide whether $\mathcal{L}(A) = \emptyset$ while the universality problem refers to the equality $\mathcal{L}(A) = \Sigma^{\omega}$. Finally, the language inclusion problem requires to decide whether $\mathcal{L}(A_0) \subseteq \mathcal{L}(A_1)$ for the given BAs A_0 and A_1. These problems have different complexity results, which are summarized by the following propositions. The corresponding proofs can be found in the cited papers or in [32, Section 4.4].

Proposition 6 (cf. [41,42,92]). *Given a BA A, the emptiness problem $\mathcal{L}(A) = \emptyset$ is decidable in linear time and is NLOGSPACE-complete.*

The proof of the linear time complexity is based on finding a strongly connected component, i.e., a set of states each one reachable from each other, which is reachable from the initial state and contains a state in F. This can be easily done by a simple graph exploration based on depth-first visit. In theory, we can also nondeterministically find an accepting state and the accepting run of A visiting the accepting state infinitely often, which is in NLOGSPACE. In fact, it is enough to guess an accepting state $q_f \in F$ and two paths: a stem path from $\bar{q}$ to q_f and a lasso path from q_f to q_f itself, both of them with length at most $|Q|$. Clearly storing $q_f \in F$ requires a space that is logarithmic in $|Q|$; for the paths, it is enough to store the current state q and a counter *cnt* to keep track of the length of the path so far; both require logarithmic space.

The algorithm works as follows: initially, $q = \bar{q}$ and the following steps are repeated to find a stem path from $\bar{q}$ to q_f: (1) from q, a successor is chosen nondeterministically and *cnt* is increased; (2) if *cnt* exceeds $|Q|$, then "no" is returned; (3) if $q = q_f$ and $\mathit{cnt} \leq |Q|$, then the algorithm turns to look for a lasso path. Starting with $q = q_f$, the following steps are repeated to find a lasso path from q_f to q_f itself: (1) from q, a successor is chosen nondeterministically and *cnt* is increased; (2) if *cnt* exceeds $|Q|$, then "no" is returned; (3) if $q = q_f$ and $\mathit{cnt} \leq |Q|$, then "yes" is returned. We refer interested reader to [41, 42, 92] for the proof of the NLOGSPACE-hardness result.

Proposition 7 (cf. [92]). *Given a BA A, the universality problem $\mathcal{L}(A) = \Sigma^{\omega}$ is decidable in exponential time and is PSPACE-complete.*

The universality problem is decided by means of a reduction to the emptiness problem: in order to decide $\mathcal{L}(A) = \Sigma^{\omega}$, it is enough to check $\mathcal{L}(A^{\mathcal{C}}) = \emptyset$, where $A^{\mathcal{C}}$ is the complement BA of A. Since $A^{\mathcal{C}}$ is exponentially larger than A, the complexity results follow from Proposition 6.

Proposition 8 (cf. [92]). *Given two BAs A_0 and A_1, the language inclusion problem $\mathcal{L}(A_0) \subseteq \mathcal{L}(A_1)$ is decidable in exponential time and is PSPACE-complete.*

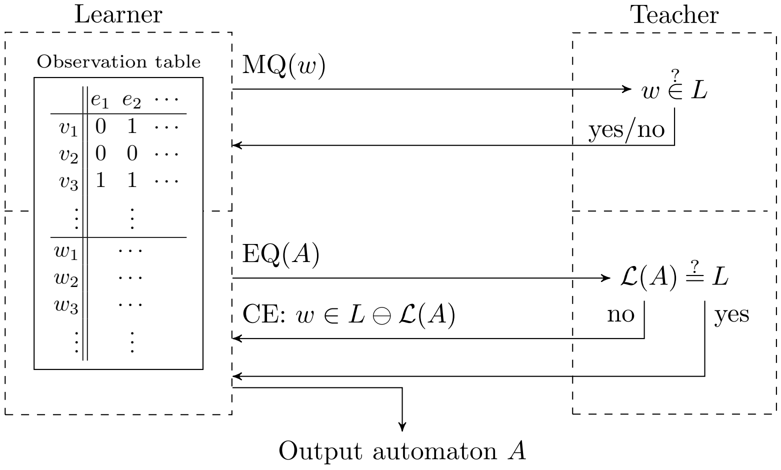


Fig. 2. DFA Active Automata Learning Framework

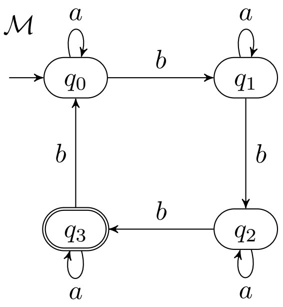
The language inclusion problem is decided by means of a reduction to the emptiness problem: in order to decide $\mathcal{L}(A_0) \subseteq \mathcal{L}(A_1)$, it is enough to check whether $\mathcal{L}(A_0) \cap \mathcal{L}(A_1^{\mathcal{C}}) = \emptyset$, where $A_1^{\mathcal{C}}$ is the complement BA of A_1. Since $A_1^{\mathcal{C}}$ is exponentially larger than A_1, the complexity results follow from Propositions 2 and 6.

4 Learning Finite Automata

In this section, we present a variant of the learning algorithm for finite automata used in [11]. In 1987, in her seminal work [8], Angluin proposed the L^* algorithm to learn a DFA accepting a target regular language; L^* belongs to the class of *active automata learning algorithms* [102], in which the learner can interact with an oracle until the correct automaton is constructed.

4.1 Overview of the DFA Learning Algorithm

As depicted in Figure 2, in the active automata learning setting presented in [8], there is a *teacher* and a *learner*. The teacher knows the target language L which can be a regular language or an ω-regular language. The learner wants to learn the target language, represented by an automaton, from the teacher by means of two kinds of queries: *membership queries* and *equivalence queries*. A membership query MQ(w) asks whether a word w belongs to L while an equivalence query EQ(A) asks whether the conjectured automaton A accepts L. Depending on whether the conjectured automaton A is correct, the teacher replies with either "yes" or "no". In case of a positive answer, the learner outputs A and completes his job. For the negative answer, the teacher provides as well a witness $w \in$



$\mathcal{T}$

	ε	bab	ab
ε	0	0	0
b	0	1	0
bb	0	0	1
bbb	1	0	0
a	0	0	0
ba	0	1	0
bba	0	0	1
$bbba$	1	0	0
$bbbb$	0	0	0

$R = \{\, u \in \{a, b\}^+ \mid \text{the number of } b \text{ in } u \text{ is } 4n + 3, \text{ for some } n \in \mathbb{N} \,\}$

Fig. 3. A DFA $\mathcal{M}$, its regular language R, and an observation table $\mathcal{T}$ for R

$L \ominus \mathcal{L}(A)$ which allows the learner to further refine the conjectured automaton A.

In this paper, the learner uses a data structure called *observation table* to store all answers to the membership queries, since it is easy to present and understand. We remark that observation tables have been originally adopted by Angluin for her L^* algorithm [8]. Instead of observation tables, the learner can use a tree-based data structure called *classification tree* to store such answers, which is usually more compact than observation tables; we refer the interested reader to [59, 63, 73] for the details on classification trees.

In the following, we play the role of the learner to learn a regular language represented by a DFA from a teacher. Regular language learning is actually a procedure for a learner to gradually identify the states in the minimal DFA $\mathcal{M}$ recognizing the target language. As an example, consider the regular language R accepted by the DFA $\mathcal{M}$ shown in Figure 3, where $R = \{\, u \in \{a, b\}^+ \mid$ the number of b in u is $4n + 3$ for some $n \in \mathbb{N} \,\}$. We observe that for any pair of words $u_1, u_2 \in \{a, b\}^*$, $\mathcal{M}(q_0, u_1) \neq \mathcal{M}(q_0, u_2)$ if there exists some word $v \in \{a, b\}^*$ such that $\mathcal{M}(q_0, u_1 v) = q_3$ while $\mathcal{M}(q_0, u_2 v) \neq q_3$. That is, in the DFA $\mathcal{M}$, for any pair of words $u_1, u_2 \in \{a, b\}^*$, if there exists some word $v \in \{a, b\}^*$ such that $u_1 v \in \mathcal{L}(\mathcal{M})$ while $u_2 v \notin \mathcal{L}(\mathcal{M})$, then $\mathcal{M}(q_0, u_1)$ and $\mathcal{M}(q_0, u_2)$ must be two different states. Our goal is to develop a learner who can identify the states in the DFA $\mathcal{M}$; using such a word extension v to distinguish words u_1 and u_2 is a good means to identify two different states in $\mathcal{M}$.

4.2 Right Congruences and Myhill-Nerode Theorem

This idea of distinguishing words by extensions is formalized by the notion of *right congruence*. A right congruence is an equivalence relation $\backsim$ on Σ^* such that $x \backsim y$ implies $xv \backsim yv$ for every $x, y, v \in \Sigma^*$. The right congruence relation is the theoretical foundation for the DFA learning algorithms to discover the states in a target DFA $\mathcal{M}$.

We denote by $|\backsim|$ the index of $\backsim$, i.e., the number of equivalence classes of $\backsim$. We use $\Sigma^*/_\backsim$ to denote the equivalence classes of the right congruence $\backsim$. A *finite right congruence* is a right congruence with a finite index. The following theorem guarantees that every regular language has a right congruence relation of finite index.

Theorem 2 (Myhill-Nerode Theorem [56]). *For a language R over Σ, the following statements are equivalent:*

1. *R is a regular language.*
2. *R is the union of some equivalence classes of a right congruence equivalence relation of finite index.*
3. *The right congruence relation $\backsim_R$ is of finite index, where $x \backsim_R y$ if and only if for each $v \in \Sigma^*$, $xv \in R \iff yv \in R$.*

The theorem basically states that given a regular language R over Σ, the whole set of finite words Σ^* can be partitioned into a finite number of equivalence classes by the right congruence relation $\backsim_R$. For a word $u \in \Sigma^*$, we denote by $[u]_\backsim$ the equivalence class of the right congruence $\backsim$ u belongs to.

Given a right congruence relation $\backsim_R$ for the language R, we can construct an automaton accepting R by means of $\backsim_R$: as set of states Q, we just use the equivalence classes induced by $\backsim_R$; the initial state $\bar{q}$ is simply the class of the empty word ε; the transition relation just considers as the a-successor of the class of u the class of ua; finally, the accepting states F are the classes of the words in R.

Definition 2 (DFA induced by $\backsim_R$). *Given a right congruence relation $\backsim_R$ for the language R, the corresponding DFA $A_{\backsim_R}$ is the tuple $A_{\backsim_R} = (Q, \bar{q}, \delta, F)$ where*

- *$Q = \Sigma^*/_{\backsim_R}$;*
- *$\bar{q} = [\varepsilon]_{\backsim_R}$;*
- *for each $u \in \Sigma^*$ and $a \in \Sigma$, $\delta([u]_{\backsim_R}, a) = [ua]_{\backsim_R}$; and*
- *$F = \{\, [u]_{\backsim_R} \in Q \mid u \in R \,\}$.*

As an example, consider the regular language R shown in Figure 3; we have four equivalence classes in $\Sigma^*/_{\backsim_R}$, namely $[\varepsilon]_{\backsim_R}$, $[b]_{\backsim_R}$, $[bb]_{\backsim_R}$, and $[bbb]_{\backsim_R}$, which intuitively correspond to how many b's have been seen so far, modulo 4; in particular, the regular language R is exactly the equivalence class $[bbb]_{\backsim_R}$. The automaton constructed from $\backsim_R$ is $\mathcal{M}$, whose states q_0, q_1, q_2, and q_3 represent the four equivalence classes $[\varepsilon]_{\backsim_R}$, $[b]_{\backsim_R}$, $[bb]_{\backsim_R}$, and $[bbb]_{\backsim_R}$, respectively.

We can use the word bab to distinguish the words in the equivalence class $[b]_{\backsim_R}$ from the words in the other three equivalence classes $[\varepsilon]_{\backsim_R}$, $[bb]_{\backsim_R}$, and $[bbb]_{\backsim_R}$. For instance, $\varepsilon \cdot bab \notin R$ while $b \cdot bab \in R$, hence $\varepsilon \not\backsim_R b$. One can check, as hinted by the column headers of the table in Figure 3, that it is enough to use the word extensions ε, bab, and ab to distinguish the words from the four equivalence classes in $\Sigma^*/_{\backsim_R}$. We can use any other word as extension, as long as it distinguishes words: for instance, we could use $(aba)^{12}$ instead of ε

or $a(ababa)^{300}b$ instead of ab. Note however that longer extensions slow down the learning algorithm, whose complexity depends also on the length of the distinguishing words (cf. Theorem 4).

Assume that we want to design a learner to learn the regular language $R = \{\, u \in \{a, b\}^{+} \mid \text{the number of } b \text{ in } u \text{ is } 4n + 3, \text{ for some } n \in \mathbb{N} \,\}$, i.e., to discover all states in the target automaton $\mathcal{M}$ as shown in Figure 3. By Theorem 2, we know that the right congruence relation $\backsim_R$, by means of word extensions, can help us to distinguish the equivalence classes of Σ^* generating R, which intuitively correspond to the states of $\mathcal{M}$. However, we do not know R and we also do not know $\backsim_R$; in order to learn them, the idea is to ask for a few words whether they belong to R, and use the obtained information to conjecture a DFA which is supposed to accept R. Yet there are still several things we are missing:

1. How does an observation table organize the results of membership queries we have collected so far?
2. How can we build a DFA from an observation table correctly?
3. How can we update an observation table and discover new states from the returned counterexample if the conjectured DFA is incorrect?

The answers to the three questions are the key cornerstones of the DFA learning algorithm. In the following, we first show how an observation table organizes the results of membership queries. Then we show how to build a DFA from an observation table. Afterwards we explain how to analyze a returned counterexample to update an observation table so to discover new states in target DFA $\mathcal{M}$. At last, we present our DFA learner for regular languages.

4.3 Observation Tables

An observation table is a tuple $\mathcal{T} = (U, V, T)$ where U is a prefix-closed set of words called *access strings*, V is a set of words called *experiments*, and $T\colon (U \cup U\Sigma)V \to \{0, 1\}$ is a total mapping.

As the name suggests, an observation table $\mathcal{T}$ is represented by a table, where rows and columns are labelled with words taken from $U \cup U\Sigma$ and V, respectively, and the table entries are the value assigned by T to them. Consider for instance the observation table $\mathcal{T} = (U, V, T)$ shown in Figure 3; the labels of the four rows in the upper part of the table correspond to the set $U = \{\varepsilon, b, bb, bbb\}$; the labels of the five rows in the bottom part of the table are those in $U\Sigma \setminus U$, so $U \cup U\Sigma$ is exactly the set of labels of the rows of the table; the labels of the three columns in the table correspond to the set $V = \{\varepsilon, bab, ab\}$. The entry value of row u and column v represents the value assigned by $T(\,\cdot\,)$ to the word uv, i.e., $T(uv)$; such a value is 1 if $uv \in R$ and 0 otherwise. As depicted in Figure 3, the entry value of row b and column bab is $T(bbab) = 1$ since $b \cdot bab \in R$, while the entry value of row b and column ab is $T(bab) = 0$ since $b \cdot ab \notin R$.

Given an observation table for a language R with right congruence $\backsim_R$, like the one in Figure 3, we can see that every equivalence class $[u]_{\backsim_R}$ of $\Sigma^*/_{\backsim_R}$ has a representative word u in U. Therefore we also use the representative word such as ε to represent the equivalence class $[\varepsilon]_{\backsim_R}$. A word in V is a word extension or experiment used to distinguish the words belonging to different equivalence classes. For instance, consider the column ab in Figure 3: the entry value at row ε is 0 while the entry value at row bb is 1, which indicates that the words from those two equivalence classes can be distinguished by the word ab. Hence any two rows with different entry values in the table are classified to be different equivalence classes while any two rows with the same entry values are seen as one equivalence class. For instance, in Figure 3, row b in the upper table and row ba in the lower table are seen as one equivalence class since they have the same entry values for each experiment.

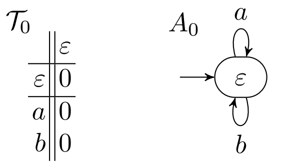
We remark that those rows which are currently seen as one equivalence class may later be classified into different equivalence classes, as result of a counterexample returned by the teacher.

The domain of the mapping T also contains the set $U\Sigma$, i.e., there are also some rows labelled by the words from $U\Sigma$ in the table. The existence of this set $U\Sigma$ of rows in the table makes it possible for the learner to look for the next equivalence class or the successor state $[ua]_{\backsim_R}$ in the DFA construction after reading a letter $a \in \Sigma$ at the equivalence class or state $[u]_{\backsim_R}$, where $u \in U$. For example, suppose that we want to compute the a-successor of state ε in Figure 3: the expected successor is $\varepsilon \cdot a$. In order to find its actual representative, we first look for the row $\varepsilon \cdot a$ in the table and then get the successor state $\varepsilon \in U$ which has the same entry values as row $\varepsilon \cdot a$. From the table we see that the row $\varepsilon \cdot a = a$ has entry values 000; the same entry values occur for the row $\varepsilon \in U$, thus the a-successor of ε is ε itself.

In order to efficiently check whether two rows represent the same equivalence class, we formally define the rows as the total function $\mathit{row} \colon (U \cup U\Sigma) \to (V \to \{0, 1\})$. To a word $u \in U \cup U\Sigma$ we assign a total function $\mathit{row}(u) \colon V \to \{0, 1\}$ such that $\mathit{row}(u)(v) = T(uv)$ for each $v \in V$. We call such a function a *row* of $\mathcal{T}$ and we denote by $\mathit{Rows}(\mathcal{T})$ the set of rows of $\mathcal{T}$; similarly, we denote by $\mathit{Rows}_{upp}(\mathcal{T}) = \{\, \mathit{row}(u) \mid u \in U \,\}$ the set of rows in the "upper" part of the table and $\mathit{Rows}_{low}(\mathcal{T}) = \{\, \mathit{row}(u) \mid u \in U\Sigma \setminus U \,\}$ the set of rows appearing in the "lower" part. Consider again the observation table in Figure 3: we have $\mathit{Rows}_{low}(\mathcal{T}) = \{\mathit{row}(a), \mathit{row}(ba), \mathit{row}(bba), \mathit{row}(bbba), \mathit{row}(bbbb)\}$, where, e.g., $\mathit{row}(a)$ is the constant function $\mathit{row}(a)(v) = 0$ for each $v \in V$. In practice, we can identify each function $\mathit{row}(u) \colon V \to \{0, 1\}$ with the content of $\mathcal{T}$ in the row labelled by u.

In the learning framework depicted in Figure 2, the learner can ask the teacher two types of queries, namely, membership and equivalence queries. In order to pose an equivalence query, the learner has to generate a DFA from the information stored in the observation table, which has to contain all information that is needed to build such a DFA. As Angluin proposed in [8], a table has such a needed information when it is *closed* and *consistent*.

A table $\mathcal{T}$ is *closed* if for any $u \in U$ and $a \in \Sigma$, there exists $u' \in U$ such that $\mathit{row}(ua) = \mathit{row}(u')$; similarly, a table is *consistent* if for any $u_1, u_2 \in U$ and $a \in \Sigma$, $\mathit{row}(u_1) = \mathit{row}(u_2)$ implies $\mathit{row}(u_1 a) = \mathit{row}(u_2 a)$. Intuitively, the


$\mathcal{T}_0$

	ε
ε	0
a	0
b	0

Fig. 4. Table $\mathcal{T}_0$ and DFA A_0

closeness of a table makes sure that every successor of a state is in the set of discovered states while the consistency of a table ensures that those words which have been classified into the same equivalence class should behave consistently, i.e., they have the same successor equivalence classes, when extended with the same letter.

We now present how the learner proceeds in the learning algorithm to learn the target language R represented by a DFA. Let the alphabet be $\Sigma = \{a, b\}$. At the beginning, the learner has no information, so he initializes both U and V to $\{\varepsilon\}$ and defines $T(uv)$ for every $u \in U \cup U\Sigma$ and $v \in V$ according to the results of membership queries, that is, he asks the teacher the membership queries MQ(ε), MQ(a), and MQ(b); the teacher answers "no" to all of them, so the learner sets $T(\,\cdot\,)$ to be the constant function 0. The result is shown as $\mathcal{T}_0$ in Figure 4. Since $\mathcal{T}_0$ is closed and consistent, we can build the DFA A_0, also depicted in Figure 4, according to Definition 3.

In case the current $\mathcal{T}$ is not closed, the learner makes it closed by repeatedly updating $\mathcal{T}$ as follows: he looks for a word $u \in U\Sigma$ such that there is no $u' \in U$ with $row(u) = row(u')$; then moves u to U and for every $a \in \Sigma$, he adds ua to $U\Sigma$ whenever needed while setting $T(uav)$ for each $v \in V$ by means of membership queries. According to [11], whenever $\mathcal{T}$ is closed, it is also consistent since by construction there do not exist $u_1, u_2 \in U$, $u_1 \neq u_2$, with $row(u_1) = row(u_2)$.

Instead of moving a single word u from $U\Sigma$ to U when $\mathcal{T}$ is not closed, the learner can also add all its prefixes Pref(u) to U just as the L^* algorithm does in [8]. This may result in a quicker growth of $\mathcal{T}$, which anyway does not change the correctness (cf. Theorem 3) and complexity (cf. Theorem 4) of the DFA learning algorithm.

4.4 DFA Construction from an Observation Table

Definition 3 answers the second question about how to build a conjecture DFA A from an observation table correctly.

Definition 3 (DFA of a Table). *Let $\mathcal{T}$ be a closed and consistent observation table. We can construct a DFA $A = (Q, \bar{q}, \delta, F)$ from $\mathcal{T}$ as follows.*

- $Q = Rows_{upp}(\mathcal{T}) = \{row(u) \mid u \in U\}$,
- $\bar{q} = row(\varepsilon)$,
- $\delta(row(u), a) = row(ua)$, *and*

- $F = \{\, row(u) \in Rows_{upp}(\mathcal{T}) \mid row(u)(\varepsilon) = 1 \,\}$.

Consider the observation table $\mathcal{T}_0$ shown in Figure 4. The learner can construct from $\mathcal{T}_0$ the DFA $A_0 = (Q_0, \bar{q}, \delta_0, F_0)$ where $Q_0 = \{row(\varepsilon) = 0\}$, $\bar{q} = row(\varepsilon)$, $F_0 = \emptyset$, and δ_0 as depicted in Figure 4. Note that in the whole paper we use the representative words $u \in U$ instead of the row functions $row(u)$ defined in Definition 3 to mark the states in a DFA; for instance, we mark the single state of A_0 with the representative word ε instead of the row function $row(\varepsilon)$. In this way, it is easier for the reader to relate the equivalence classes to the states in the conjectured automaton.

Now the conjectured DFA A_0 is constructed and the learner can pose the equivalence query $\mathrm{EQ}(A_0)$ to the teacher. A_0 is clearly not the right conjecture, so the teacher answers "no" together with a counterexample, say $bbab \in \mathcal{L}(\mathcal{M}) \ominus \mathcal{L}(A_0)$. In the following we provide the answer to the third question, that is, how to update the observation table from the received counterexample.

4.5 Counterexample Analysis

On receiving the counterexample w, the learner has to analyze w in order to update the observation table; this would then allow the learner to expand the conjectured DFA by adding new states to correctly classify the received counterexample. To discover new states in $\mathcal{M}$, we essentially need new experiments for the table; the following lemma provides a way to find such new experiments.

Lemma 1. *Let R be the target language and A be the conjectured DFA. On receiving a counterexample $v \in R \ominus \mathcal{L}(A)$, we can always find an experiment $v' \in Suf(v)$, words $u, u' \in U$, and letter $a \in \Sigma$ such that $row(ua) = row(u')$ and $uav' \in R \iff u'v' \notin R$.*

As a notation, we use $\mathrm{MQ}(s, w)$ to denote the membership query $\mathrm{MQ}(s \cdot w)$ in order to give a clear presentation of the analysis procedure on the returned counterexample v as explained in the following. On receiving a counterexample $v \in R$ and $v \notin \mathcal{L}(A)$, the learner can check whether the membership queries return different results for v and $\tilde{v}$ where $\tilde{v} = A(v)$. Let $n = |v|$ and for $i \in [1 \cdots n]$, let $s_i = A(v[1..i])$ be the state reached after reading the first i letters of v. Recall that $s_i \in U$ is the representative word of that state in the upper part of the observation table. In particular, $s_0 = \varepsilon$. Therefore, $\tilde{v} = s_n$ and there is a sequence of membership queries $\mathrm{MQ}(s_0, v[1..n] = v)$, $\mathrm{MQ}(s_1, v[2..n])$, $\mathrm{MQ}(s_2, v[3..n])$, and so on, up to $\mathrm{MQ}(s_n, \varepsilon) = \mathrm{MQ}(\tilde{v}, \varepsilon)$. This sequence has different results for the first and the last query since $s_0 \cdot v \in R$ while $\tilde{v} \cdot \varepsilon \notin R$ by the assumption. It follows that there exists an experiment $v[i+1..n]$ for the earliest $1 \leq i \leq n$ distinguishing $s_{i-1}a'$ from s_i. Let $u = s_{i-1}$, $u' = s_i$, $a = a'$, and $v' = v[i+1..n]$. According to Definition 3, we have $row(ua) = row(u')$ since $A(s_{i-1}a) = A(ua) = u' = s_i$ and $uav' \in R$ while $u'v' \notin R$. The handling for the other case when $v \notin R$ and $v \in \mathcal{L}(A)$ is symmetric.

According to Lemma 1, on receiving a counterexample $bbab \in R \ominus \mathcal{L}(A_0)$, the learner poses a sequence of membership queries $\mathrm{MQ}(s_0 = \varepsilon, bbab)$, $\mathrm{MQ}(s_1 =$

$\mathcal{T}_0'$

	ε	bab
ε	0	0
a	0	0
b	0	1

$\mathcal{T}_1$

	ε	bab
ε	0	0
b	0	1
a	0	0
ba	0	1
bb	0	0

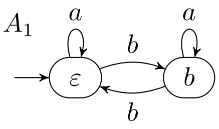


Fig. 5. Tables $\mathcal{T}_0'$, $\mathcal{T}_1$, and DFA A_1

ε, bab), MQ($s_2 = \varepsilon, ab$), MQ($s_3 = \varepsilon, b$), and MQ($s_4 = \varepsilon, \varepsilon$); it is easy to check that the experiment $v[2..4] = bab$ distinguishes $s_0 b = b$ from $s_1 = \varepsilon$. Therefore, the learner adds bab into the set V and updates the mapping T via membership queries, until obtaining the observation table $\mathcal{T}_0'$ shown in Figure 5. As $\mathcal{T}_0'$ is not closed since there is no $u \in U$ such that $row(u) = row(b)$, the learner moves the row b to the upper table, i.e., to the set U, and adds the rows ba and bb —the one letter extensions of b— to the lower part of the table as mentioned before. The learner then fills the missing entry values by means of membership queries; the resulting observation table is $\mathcal{T}_1$ shown in Figure 5. As $\mathcal{T}_1$ is closed and also consistent, the learner can build the DFA A_1 from $\mathcal{T}_1$, depicted in Figure 5.

We remark that instead of finding just one experiment $v' \in \mathrm{Suf}(v)$, our learner may also add all its suffixes $\mathrm{Suf}(v)$ into V just as the algorithm does in [76]. This may also result in a quicker grown of $\mathcal{T}$, which anyway does not change the correctness (cf. Theorem 3) and complexity (cf. Theorem 4) of the DFA learning algorithm we are presenting.

The learner poses now the equivalence query EQ(A_1) to the teacher; since $\mathcal{L}(A_1) \neq R$, the teacher returns “no” and a counterexample, say again $bbab \in \mathcal{L}(\mathcal{M}) \ominus \mathcal{L}(A_1)$. Similarly to the previous counterexample analysis, the learner asks the sequence of membership queries MQ($s_0 = \varepsilon, bbab$), MQ($s_1 = b, bab$), MQ($s_2 = \varepsilon, ab$), MQ($s_3 = \varepsilon, b$), and MQ($s_4 = \varepsilon, \varepsilon$), which allows the learner to find the experiment $w[3..4] = ab$ to distinguish $s_1 b = bb$ from $s_2 = \varepsilon$. The learner adds ab into the set V and updates T by further membership queries, resulting in the observation table $\mathcal{T}_1'$ shown in Figure 6.

$\mathcal{T}_1'$ is not closed, since there is no row in the upper part corresponding to $row(bb)$, so the learner moves bb to the upper part, adds bba and bbb to the lower part, and fills the content of the table by means of membership queries. The result of these operations is table $\mathcal{T}_1''$ which is still not closed since there does not exist $u \in U$ such that $row(bbb) = row(u)$. Therefore, as before, the learner moves bbb to the upper part, adds the missing words $bbba$ and $bbbb$ to the lower part, and fills the content, obtaining the table $\mathcal{T}_2$ depicted in Figure 6, which is now closed. The DFA A_2 constructed from $\mathcal{T}_2$ is depicted in Figure 6 and the learner gets the answer “yes” from the teacher after posing the equivalence query EQ(A_2), which means that he has completed his learning task.

$\mathcal{T}_1'$

	ε	bab	ab
ε	0	0	0
b	0	1	0
a	0	0	0
ba	0	1	0
bb	0	0	1

$\mathcal{T}_1''$

	ε	bab	ab
ε	0	0	0
b	0	1	0
bb	0	0	1
a	0	0	0
ba	0	1	0
bba	0	0	1
bbb	1	0	0

$\mathcal{T}_2$

	ε	bab	ab
ε	0	0	0
b	0	1	0
bb	0	0	1
bbb	1	0	0
a	0	0	0
ba	0	1	0
bba	0	0	1
$bbba$	1	0	0
$bbbb$	0	0	0

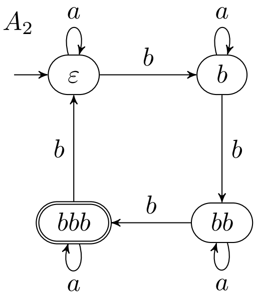


Fig. 6. Tables $\mathcal{T}_1'$, $\mathcal{T}_1''$, $\mathcal{T}_2$, and DFA A_2

Algorithm 1: The DFA Learner

```
Initialize table $\mathcal{T} = (T, U, V)$ with $U = \{\varepsilon\}$ and $V = \{\varepsilon\}$;
$CloseTable(\mathcal{T}, \mathrm{MQ}(\cdot))$ and let $A = Aut(\mathcal{T})$;
Let $(a, v)$ be the teacher's response on $\mathrm{EQ}(A)$;
while $a =$ "no" do
    $V = V \cup FindDistinguishingExperiment(v)$;
    $CloseTable(\mathcal{T}, \mathrm{MQ}(\cdot))$ and let $A = Aut(\mathcal{T})$;
    Let $(a, v)$ be the teacher's response on $\mathrm{EQ}(A)$;
return $A$;
```

4.6 The Learner

In the previous part of this section we have introduced a regular language learning algorithm by means of a running example. We now give the formal definition of the learner by means of Algorithm 1 for completeness of presentation; we can see that it agrees with the learning procedure we presented above. The function *CloseTable* is responsible for closing a table $\mathcal{T}$, so it needs to perform membership queries $\mathrm{MQ}(\cdot)$ to fill the missing entry values in $\mathcal{T}$. Moreover, as we have seen, it may repeatedly move rows from the lower to the upper part of the input table and add new rows to the lower part, until the table becomes closed. All conjectured DFAs are constructed from the table $\mathcal{T}$ by calling the function $Aut(\mathcal{T})$ based on Definition 3. On receiving a counterexample v, the function $FindDistinguishingExperiment(v)$ gets a new experiment which is later added into the set V. The refinement loop of the conjecture A terminates once we get a positive answer from the teacher.

The soundness and completeness of Algorithm 1 is guaranteed by Theorem 3.

Theorem 3. *Assume that R is the target regular language. Algorithm 1 terminates and returns a DFA A such that $\mathcal{L}(A) = R$.*

The returned DFA A from Algorithm 1 is a correct conjecture automaton simply because the teacher has approved it. The remaining problem is how we show the termination of Algorithm 1. The reason why Algorithm 1 terminates is that: 1) by Lemma 1, we can discover new states, i.e., new equivalence classes in $\Sigma^*/_{\backsim_R}$, whenever receiving a counterexample from the teacher and 2) the index of $\backsim_R$ is finite according to Theorem 2. It follows immediately the complexity result in Theorem 4.

Theorem 4. *Let R be the target regular language and $n = |\backsim_R|$; let m be the maximum length of any counterexample returned by the teacher.*

1. *Algorithm 1 terminates on receiving at most n counterexamples.*
2. *The number of membership queries is in $\mathcal{O}(n^2 \cdot |\Sigma| + n \cdot m)$.*

5 Learning Büchi Automata

After presenting the DFA learning algorithm in Section 4, we are now ready to introduce the learning algorithm for Büchi automata. Throughout this section, except stated otherwise, we let the ω-regular language L be the target language.

We have seen that, for learning a regular language R, the right congruence relation $\backsim_R$ plays an important role in identifying the equivalence classes in $\Sigma^*/_{\backsim_R}$, so we could consider to extend such an approach to the ω-regular language setting. It would be easy to learn ω-regular languages by means of BAs if we can characterize them by a right congruence relation $\backsim_L$ of finite index for each given ω-regular language L. There are, however, few questions to be answered for such an extension:

- How can we use finite memory to represent an ω-word, which has infinite length?
- Is there a right congruence relation $\backsim_L$ of finite index for a given ω-regular language L?

The answer to the first question is easy: we only need to learn the set of ultimately periodic words UP(L) for a given ω-regular language L, since by Theorem 1 the set UP(L) is the fingerprint of L; given that every ultimately periodic word w can be written as a pair of finite words (u, v) with $w = uv^\omega$, only finite memory is needed for storing w.

5.1 Right Congruences for ω-Regular Languages

In contrast, the answer to the second question is more tricky: a first proposal for extending the right congruence relation $\backsim_R$ with respect to the ω-regular language L replaces the extension $v \in \Sigma^*$ with the ultimately periodic extension xy^ω for $x \in \Sigma^*$ and $y \in \Sigma^+$.

Definition 4. *Let u_1 and u_2 be words in Σ^*. $u_1 \backsim_L u_2$ if and only if for every $x \in \Sigma^*$ and $y \in \Sigma^+$, $u_1xy^\omega \in L \iff u_2xy^\omega \in L$.*

Based on the right congruence $\backsim_L$, Maler and Pnueli [76] introduced a learning algorithm to learn a strict subset of ω-regular languages. Nonetheless, the right congruence relation $\backsim_L$ is in general not enough to learn an ω-regular language L, as the following example shows.

Example 2. Assume $L = \{a, b\}^* \cdot b^\omega$. The index of $\backsim_L$ is 1 and the only equivalence class is $[\varepsilon]_{\backsim_L}$. This follows from the fact that for any $u \in \Sigma^*$, we have $u \cdot xy^\omega \in L$ if $y^\omega = b^\omega$, otherwise $u \cdot xy^\omega \notin L$. Therefore, we only have one state with self-loops in the conjectured BA A which certainly does not recognize the target language L, since A accepts either Σ^ω or $\emptyset$, depending on whether the single state is accepting or not, respectively.

The reason why it is so difficult to learn ω-regular languages via Büchi automata is that there is a lack of right congruence for Büchi automata compared to DFAs and regular languages. Farzan *et al.* in [43] proposed the first learning algorithm to learn the complete class of ω-regular languages by means of Büchi automata; their algorithm circumvents the lack of right congruence by first using L^* to learn the DFA $D_\$$, as defined in [26], and then transforming $D_\$$ to a BA. Basically, the DFA $D_\$$ captures the set of ultimately periodic words of L by means of the regular language $\mathcal{L}(D_\$) = \{\, u\$v \mid u \in \Sigma^*, v \in \Sigma^+, uv^\omega \in L \,\}$, where $\$ \notin \Sigma$.

Another way to solve the lack of right congruence is to define a Myhill-Nerode like theorem for ω-regular languages. Inspired by the work of Arnold [12], Maler and Stager [77] proposed the notion of *family of right-congruences* (FORC for short) and presented a "Myhill-Nerode" theorem for ω-languages. The idea underlying the definition of FORC is based on the fact that every ω-regular language L can be written in the form of an ω-regular expression $\bigcup_{i=1}^{n} U_i \cdot V_i^\omega$ for some $n \in \mathbb{N}$, where for any $i \in [1 \cdots n]$, U_i and V_i are regular languages. So the intuition of using FORC is to first define a right congruence $\backsim$ to distinguish all finite word prefixes, and then define a right congruence $\approx_u$ for the finite word periods for each equivalence class $[u]_\backsim$ of the finite word prefixes. Hence we see that $[u_i]_\backsim = U_i$ with $u_i \in U_i$ being an equivalence class in $\Sigma^*/_\backsim$ and $[v_i]_{\approx_{u_i}} = V_i$ being an equivalence class in $\Sigma^*/_{\approx_{u_i}}$ such that $u_i \cdot V_i^\omega \subseteq L$.

5.2 Family of Deterministic Finite Automata

Based on this idea of FORC, Angluin and Fisman [11] recently proposed to learn ω-regular languages via a formalism called *family of DFAs* (FDFA for short), in which every DFA corresponds to a right congruence of finite index. Further, Angluin *et al.* [9] suggest to use FDFAs as language acceptors of ω-regular languages. The BA learning algorithm described in this section first learns an FDFA and then transforms it to a BA. The formal definition of an FDFA is as follows.

Definition 5 (Family of DFAs [9]). *A family of DFAs $\mathcal{F} = (M, \{A^q\})$ consists of a leading DFA $M = (Q, \bar{q}, \delta, \emptyset)$ and a set of progress DFAs $\{\, A^q = (Q_q, \bar{q}_q, \delta_q, F_q) \mid q \in Q \,\}$.*

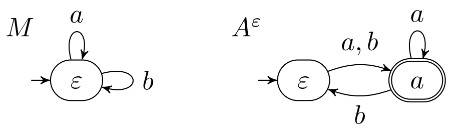


Fig. 7. An example of an FDFA $\mathcal{F} = (M, \{A^{\varepsilon}\})$

An example of FDFA $\mathcal{F}$ is depicted in Figure 7 where the leading DFA M has only one state ε and the progress DFA corresponding to the state ε is A^{ε}.

Each FDFA $\mathcal{F}$ characterizes a set of ultimately periodic words $\mathrm{UP}(\mathcal{F})$ by the acceptance condition defined as follows.

Definition 6 (Acceptance condition of FDFA). *Let $\mathcal{F} = (M, \{A^q\})$ be a FDFA and w be an ultimately periodic word. We say that*

- *w is accepted by $\mathcal{F}$ if there exists a decomposition (u, v) of w accepted by $\mathcal{F}$;*
- *a decomposition (u, v) is accepted by $\mathcal{F}$ if $M(uv) = M(u)$ and the decomposition (u, v) is captured by $\mathcal{F}$; and*
- *a decomposition (u, v) is captured by $\mathcal{F}$ if $v \in \mathcal{L}(A^q)$ where $q = M(u)$.*

Consider the FDFA $\mathcal{F}$ in Figure 7: $(ab)^{\omega}$ is accepted by $\mathcal{F}$ since there exists the decomposition (a, ba) of $(ab)^{\omega}$ such that $M(a \cdot ba) = M(a) = \varepsilon$ and $ba \in \mathcal{L}(A^{M(a)}) = \mathcal{L}(A^{\varepsilon})$. Note that the decomposition (ab, ab) of $(ab)^{\omega}$ is not accepted by $\mathcal{F}$ since (ab, ab) is not captured by $\mathcal{F}$, i.e., $ab \notin \mathcal{L}(A^{M(ab)}) = \mathcal{L}(A^{\varepsilon})$.

In the following, we recall the definition of the complement of an FDFA $\mathcal{F}$.

Definition 7 (Complement of FDFA [9]). *Given an FDFA $\mathcal{F} = (M, \{A^q\})$, the* complement *$\mathcal{F}^{\mathcal{C}}$ of $\mathcal{F}$ is the FDFA $\mathcal{F}^{\mathcal{C}} = (M, \{(A^q)^{\mathcal{C}}\})$.*

It is easy to see that the complement FDFA $\mathcal{F}^{\mathcal{C}}$ captures every decomposition (u, v) in $\Sigma^* \times \Sigma^+$ which is not captured by $\mathcal{F}$.

It is shown in [11] that for every ω-regular language L, there exists an FDFA $\mathcal{F}$ such that $\mathrm{UP}(\mathcal{F}) = \mathrm{UP}(L)$. More precisely, Angluin and Fisman [11] suggest to use three kinds of FDFAs as canonical representations of ω-regular languages, namely *periodic FDFAs*, *syntactic FDFAs*, and *recurrent FDFAs*. In this work, we only consider the periodic FDFAs to simplify the presentation of the BA learning algorithm; we refer the interested reader to [11, 73] for more details on the other two canonical FDFAs.

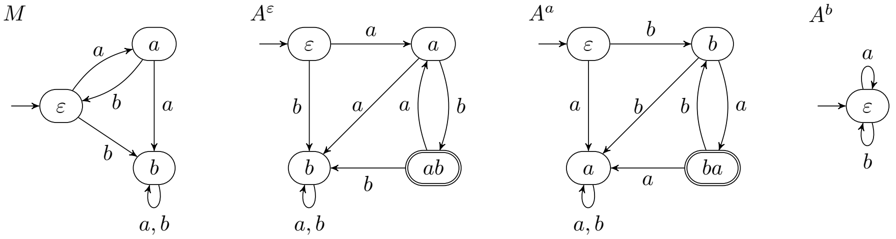
The definition of periodic FDFAs provided in [11] is given in terms of right congruences.

Definition 8 (Periodic FDFA [11]). *Let L be an ω-regular language.*

Given $u \in \Sigma^*$, the periodic right congruence *$\approx_P^u$ is an equivalence relation on Σ^* such that for each $x, y \in \Sigma^*$, $x \approx_P^u y$ if and only if for each $v \in \Sigma^*$, it holds $u(xv)^{\omega} \in L \iff u(yv)^{\omega} \in L$.*

The periodic FDFA $\mathcal{F}$ of L is the FDFA $\mathcal{F} = (M, \{A^u\})$ where:

- *the DFA $M = (\Sigma^*/_{\backsim_L}, [\varepsilon]_{\backsim_L}, \delta, \emptyset)$ is the* leading DFA*, where $\delta([u]_{\backsim_L}, a) = [ua]_{\backsim_L}$ for each $u \in \Sigma^*$ and $a \in \Sigma$;*



$\mathcal{T}$

$\backsim_L$	(ε, ab)	(ε, ba)
ε	1	0
a	0	1
b	0	0
aa	0	0
ab	1	0
ba	0	0
bb	0	0

$\mathcal{T}_\varepsilon$

$\approx_P^\varepsilon$	ε	ab	b
ε	0	1	0
a	0	0	1
b	0	0	0
ab	1	1	0
aa	0	0	0
ba	0	0	0
bb	0	0	0
aba	0	0	1
abb	0	0	0

$\mathcal{T}_a$

$\approx_P^a$	ε	ba	a
ε	0	1	0
a	0	0	0
b	0	0	1
ba	1	1	0
aa	0	0	0
ab	0	0	0
bb	0	0	0
baa	0	0	0
bab	0	0	1

$\mathcal{T}_b$

$\approx_P^b$	ε
ε	0
a	0
b	0

Fig. 8. A periodic FDFA $\mathcal{F} = (M, \{A^\varepsilon, A^a, A^b\})$ with $\mathrm{UP}(\mathcal{F}) = (ab)^\omega$

- *for each* $[u]_{\backsim_L} \in \Sigma^*/_{\backsim_L}$*, the DFA* $A^u = (\Sigma^*/_{\approx_P^u}, [\varepsilon]_{\approx_P^u}, \delta_u, F_u)$ *is a* progress DFA*, where* $\delta_u([v]_{\approx_P^u}, a) = [va]_{\approx_P^u}$ *for each* $v \in \Sigma^*$ *and* $a \in \Sigma$*, and* $F_u = \{\, [v]_{\approx_P^u} \in \Sigma^*/_{\approx_P^u} \mid uv^\omega \in L \,\}$.

As shown in [11], given an ω-regular language L, $\backsim_L$ and $\approx_P^u$ for any $u \in \Sigma^*$ are all right congruences of finite index, so DFAs can be built from them.

We remark that the set of ultimately periodic words $\mathrm{UP}(\mathcal{F})$ accepted by the periodic FDFA $\mathcal{F}$ is consistent with those characterized by the regular language $\mathcal{L}(D_\$)$ defined in [26].

Consider the periodic FDFA $\mathcal{F}$ depicted in Figure 8 where $\mathcal{F}$ characterizes the ω-regular language $L = (ab)^\omega$. The leading DFA M of $\mathcal{F}$ has three states, namely ε, a, and b which correspond to the equivalence classes $[\varepsilon]_{\backsim_L}$, $[a]_{\backsim_L}$, and $[b]_{\backsim_L}$, respectively, given by the upper part U of $\mathcal{T}$. The set of experiments V contains decompositions of ultimately periodic words, which play the role of x and y in Definition 4 introducing $\backsim_L$. As shown in table $\mathcal{T}$ for M, the experiments (ε, ab) and (ε, ba) are enough to distinguish the equivalence classes in $\Sigma^*/_{\backsim_L}$. In fact, for any $u_1, u_2 \in \Sigma^*$, an experiment xy^ω for which $xy^\omega \neq ((ab)^+)^\omega$ and $xy^\omega \neq ((ba)^+)^\omega$ cannot distinguish u_1 and u_2, since for sure we have $u_1 \cdot xy^\omega \notin L$ and $u_2 \cdot xy^\omega \notin L$. Since the leading DFA has three states ε, a, and b, there are three progress DFAs in $\mathcal{F}$ associated to them, namely A^ε, A^a, and A^b, respectively.

In the following, we show how the periodic FDFA $\mathcal{F}$ corresponds to the ω-regular expression $\varepsilon(ab)^* \cdot ((ab)^+)^\omega \cup a(ba)^* \cdot ((ba)^+)^\omega \cup \{b, a(ba)^*a\}\{a, b\}^* \cdot \emptyset$, where for clarity of presentation we use $\cup$ instead of the usual symbol $+$ to distinguish the different ω-regular expressions.

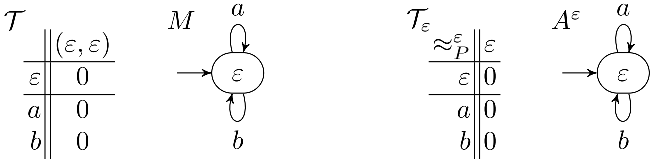
Let us consider the first part of the ω-regular expression, i.e., $\varepsilon(ab)^* \cdot ((ab)^+)^\omega$, which corresponds to the state ε of M and its progress DFA A^ε, whose language is clearly $\mathcal{L}(A^\varepsilon) = (ab)^+$. Let the components of M be $(Q, \varepsilon, \delta, \emptyset)$ and consider the DFA $M^\varepsilon = (Q, \varepsilon, \delta, \{\varepsilon\})$ obtained by setting the accepting set of M to $\{\varepsilon\}$. It is easy to see that $\mathcal{L}(M^\varepsilon) = \varepsilon(ab)^*$. Let $U_1 = \mathcal{L}(M^\varepsilon) = \varepsilon(ab)^*$ and $V_1 = \mathcal{L}(A^\varepsilon) = (ab)^+$. It follows that the expression $U_1 \cdot V_1^\omega$ is exactly $\varepsilon(ab)^* \cdot ((ab)^+)^\omega$. Similarly we can get the other two ω-regular expressions $a(ba)^* \cdot ((ba)^+)^\omega$ and $\{b, a(ba)^*a\}\{a,b\}^* \cdot \emptyset$ from the remaining two states a and b of M and their corresponding progress DFAs A^a and A^b, respectively. Note that $\varepsilon(ab)^* \cdot ((ab)^+)^\omega \cup a(ba)^* \cdot ((ba)^+)^\omega \cup \{b, a(ba)^*a\}\{a,b\}^* \cdot \emptyset = (ab)^\omega$, that is, the induced ω-regular expression corresponds to the language accepted by $\mathcal{F}$. In general, we can construct from the periodic FDFA $\mathcal{F}$ accepting L a unique ω-regular expression representing L.

We remark that by fixing a state of M, say state ε, the right congruence $\approx_P^\varepsilon$ is actually the same as the right congruence $\backsim_R$ for the regular language $R = V_1$. Recall that the idea underlying FORC is to first define a right congruence $\backsim$ distinguishing all finite word prefixes, and then define, for each equivalence class $[u]_\backsim$ of the finite word prefixes, a right congruence $\approx_u$ for the finite word periods. Thus after fixing the equivalence class $[\varepsilon]_{\backsim_L}$ for the finite word prefixes of L, we can define the right congruence $\backsim_R$ for the regular language $R = V_1$ of finite word periods defined as $\{\, v \in \{a,b\}^+ \mid \varepsilon \cdot v^\omega \in L \,\}$ and call it $\approx_P^\varepsilon$. These right congruences allow for the development of a learning algorithm for ω-regular languages represented by FDFAs, where the FDFA learner can be seen as a procedure to simultaneously run an instance of the DFA learner for each DFA in the FDFA. In the remaining part of this section we first introduce a periodic FDFA learner and then present the learning algorithm for BAs.

5.3 Learning a Family of DFAs

In order to present the periodic FDFA learner, we need first introduce the observation tables for each internal DFA learner. In this work, we often use FDFA learner as a shorthand for the periodic FDFA learner since we only consider periodic FDFAs. We remark that the FDFA learner introduced in this section is specialized for the periodic FDFAs which differs from the FDFA learner specified in [11, 73] by requiring the received counterexamples satisfying Definition 9.

Observation Tables for a Family of DFAs. An observation table $\mathcal{T}$ for the leading DFA learner, called *leading table*, has the same structure (U, V, T) as the one for the DFA learner presented in Section 4.3 except that T and V are adapted to handle ω-regular words: V is a set of decompositions rather than a set of finite words; $T \colon (U \cup U\Sigma)V \to \{0, 1\}$ is still a mapping but the entry value of row u and column (x, y), denoted by $T(u, (x, y))$, is 1 if $uxy^\omega \in L$ and 0 otherwise. Consider for instance the leading table $\mathcal{T}$ shown in Figure 8: we have that $V = \{(\varepsilon, ab), (\varepsilon, ba)\}$ is the set of experiments and $T(a, (\varepsilon, ab)) = 0$ since $a \cdot \varepsilon \cdot (ab)^\omega \notin L$ while $T(a, (\varepsilon, ba)) = 1$ since $a \cdot \varepsilon \cdot (ba)^\omega \in L$. The *row* function



$\mathcal{T}$	$(\varepsilon, \varepsilon)$
ε	0
a	0
b	0

$\approx_P^\varepsilon$	ε
ε	0
a	0
b	0

Fig. 9. The initial FDFA $\mathcal{F}_0$ and its corresponding tables while learning $(ab)^\omega$

remains unchanged, thus we still have $\mathit{row}\colon (U \cup U\Sigma) \to (V \to \{0,1\})$ being a total function such that for each word $u \in U \cup U\Sigma$, $\mathit{row}(u)\colon V \to \{0,1\}$ is a total function defined as $\mathit{row}(u)(x,y) = T(u,(x,y))$ for each $(x,y) \in V$.

For every $u \in U$ of the leading table $\mathcal{T}$, there exists an observation table $\mathcal{T}_u$ for the progress DFA learner called *progress table*. $\mathcal{T}_u$ has the same structure (U_u, V_u, T_u) as the one for the DFA learner (cf. Section 4.3) except that the entry value of row x and column v, denoted by $T_u(x,v)$, is 1 if $u \cdot (xv)^\omega \in L$ and 0 otherwise. Consider for instance the table $\mathcal{T}_a$ shown in Figure 8: $T_a(\varepsilon, ba) = 1$ since $a \cdot (\varepsilon ba)^\omega \in L$ while $T_a(\varepsilon, a) = 0$ since $a \cdot (\varepsilon a)^\omega \notin L$.

Unless stated otherwise, all remaining notions for the table of a DFA learner can be also directly applied to the leading table and progress tables, such as the DFA construction from a table and the closeness and consistency of a table.

The Learning Procedure of the FDFA Learner After the introduction of the observation tables for the FDFA learner, we are now ready to give the intuition about how the FDFA learner works by learning the ω-regular language $L = (ab)^\omega$ over $\Sigma = \{a,b\}$.

As for the DFA learner, at the beginning the FDFA learner has no information so he initializes the components U and V of the leading table $\mathcal{T}$ to $\{\varepsilon\}$ and $\{(\varepsilon,\varepsilon)\}$, respectively. Then he turns to fill the content of T, so for each $u \in U \cup U\Sigma$ and $(x,y) \in V$, he makes a membership query $\mathrm{MQ}(u \cdot x, y)$ whose answer is stored as $T(u,(x,y))$; the membership query $\mathrm{MQ}(f,g)$ is used to asks the FDFA teacher whether the word fg^ω belongs to L.

Once T is fully defined, the learner checks whether $\mathcal{T}$ is closed; if it is not closed, he repeatedly moves rows from the lower part to the upper part, adds the new rows in $U\Sigma$ as needed, and fills T, as done by the DFA learner (see Section 4.3), until $\mathcal{T}$ becomes closed.

As soon as $\mathcal{T}$ is closed, the learner constructs the corresponding leading DFA M and then turns to the progress tables: for each $u \in U$ of $\mathcal{T}$, he first creates a progress table $\mathcal{T}_u$ and then initializes both U_u and V_u to $\{\varepsilon\}$. For every $x \in U_u \cup U_u\Sigma$ and $v \in V_u$, $T_u(x,v)$ is defined according to the result of the membership query $\mathrm{MQ}(u, xy)$. Then the learner makes sure that each progress table $\mathcal{T}_u$ is closed before constructing the corresponding progress DFA A^u. Once all DFAs are constructed, he is ready to pose the first equivalence query $\mathrm{EQ}(\mathcal{F}_0)$ for the conjectured $\mathcal{F}_0$ to the FDFA teacher; $\mathcal{F}_0$ is shown in Figure 9 together with its corresponding tables.

On receiving EQ($\mathcal{F}$), the teacher has to decide whether the conjectured FDFA $\mathcal{F}$ is an appropriate periodic FDFA of the target language L. $\mathcal{F}_0 = (M, \{A^\varepsilon\})$ is clearly not the right conjecture so she answers "no" and provides a counterexample, say the decomposition (ε, ab). Note that the counterexample (x, y) returned by the teacher is not just an ultimately periodic word $xy^\omega \in \mathrm{UP}(\mathcal{F}) \ominus \mathrm{UP}(L)$, but it needs to satisfy additional requirements given in the following definition, in order to be useful for the learner to refine the conjectured FDFA.

Definition 9 (Counterexample for the FDFA learner). *Let L be the target language and $\mathcal{F}$ be the conjectured FDFA. We say that a counterexample (u, v) is*

- positive *if (u, v) is not captured by $\mathcal{F}$ and $uv^\omega \in UP(L)$, and*
- negative *if (u, v) is captured by $\mathcal{F}$ and $uv^\omega \notin UP(L)$.*

Remark 1. Besides the periodic FDFAs, Angluin and Fisman [11] introduced also the recurrent and the syntactic FDFAs, which make use of a different definition of right congruence. Similarly to the periodic case, also for these two FDFAs it is possible to define positive and negative counterexamples, which are however more involved. We refer the interested reader to [73] for more details on these two other types of FDFAs.

Refinement of the Conjectured FDFA $\mathcal{F}$. In order to decide which DFA in the conjectured $\mathcal{F}$ has to be refined, the learner acts differently depending on whether the received counterexample (u, v) is positive or negative.

If (u, v) is a positive counterexample, the learner proceeds as follows: let $\tilde{u} = M(u)$; if $\tilde{u} \cdot v^\omega \in \mathrm{UP}(L)$, then the progress DFA $A^{\tilde{u}}$ is refined, otherwise the leading DFA M is refined. In case (u, v) is a negative counterexample, the learner just acts symmetrically: if $\tilde{u} \cdot v^\omega \in \mathrm{UP}(L)$, then M is refined, otherwise $A^{\tilde{u}}$ is refined.

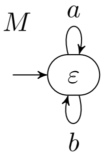
Consider again the conjectured FDFA $\mathcal{F}$ shown in Figure 9 and the returned counterexample (ε, ab): (ε, ab) is clearly a positive counterexample so the conjectured progress DFA A^ε has to be refined since $\tilde{u} = \varepsilon = M(\varepsilon)$ and $\varepsilon \cdot (ab)^\omega \in \mathrm{UP}(L)$.

Refinement of the progress DFA $A^{\tilde{u}}$. Assume that (u, v) is a positive counterexample: by definition we have that $\tilde{u} \cdot v^\omega \in \mathrm{UP}(L)$ and $A^{\tilde{u}}$ has to be refined so to accept v.

The counterexample analysis is similar to Lemma 1 due to the close relation of $\approx_P^u$ with $\backsim_R$: let $n = |v|$ and for each $i \in [1 \cdots n]$, let $s_i = A^{\tilde{u}}(v[1..i])$ be the state in $A^{\tilde{u}}$ after reading the first i letters of v; recall that $s_0 = \varepsilon$. There exists a sequence of membership queries MQ($\tilde{u}, s_0 \cdot v[1..n]$), MQ($\tilde{u}, s_1 \cdot v[2..n]$), and so on, up to MQ($\tilde{u}, s_n \cdot \varepsilon$). By assumption we have $\tilde{u} \cdot (s_0 \cdot v[1..n])^\omega \in L$ while $\tilde{u} \cdot (s_n \cdot \varepsilon)^\omega \notin L$ due to the fact that (u, v) is not captured by $\mathcal{F}$ and thus s_n is not an accepting state. Recall that by Definition 8, the accepting set $F_{\tilde{u}}$ of the progress DFA $A^{\tilde{u}}$ is the set of equivalence classes $\{\, [v]_{\approx_P^{\tilde{u}}} \mid \tilde{u} \cdot v^\omega \in$

$\mathcal{T}$

	$(\varepsilon, \varepsilon)$
ε	0
a	0
b	0



$\mathcal{T}_\varepsilon$

$\approx^\varepsilon_P$	ε	b
ε	0	0
a	0	1
ab	1	0
b	0	0
aa	0	0
aba	0	1
abb	0	0

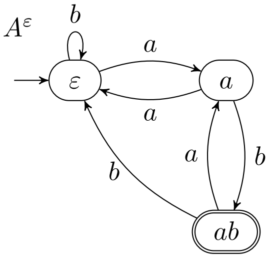


Fig. 10. The refined FDFA $\mathcal{F}_1$ and its corresponding tables while learning $(ab)^\omega$

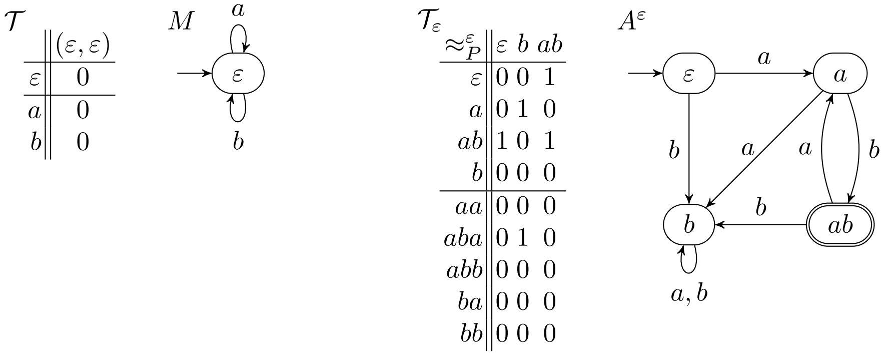
$L\}$. Therefore, the learner can find the first experiment $v[j+1..n]$ such that $\tilde{u} \cdot (s_{j-1}v[j] \cdot v[j+1..n])^\omega \in L$ while $\tilde{u} \cdot (s_j \cdot v[j+1..n])^\omega \notin L$; this means that $s_{j-1}v[j]$ and s_j do not represent the same equivalence class and must be split.

Consider again the conjectured FDFA $\mathcal{F}_0$ shown in Figure 9 that has to be refined by means of the positive counterexample (ε, ab), which requires to refine the progress A^ε: from the sequence of membership queries $\mathrm{MQ}(\varepsilon, \langle s_0 = \varepsilon\rangle \cdot ab)$, $\mathrm{MQ}(\varepsilon, \langle s_1 = \varepsilon\rangle \cdot b)$, and $\mathrm{MQ}(\varepsilon, \langle s_2 = \varepsilon\rangle \cdot \varepsilon)$, the learner finds the experiment b distinguishing $\varepsilon \cdot a$ from ε. So he first adds b to V_ε of table $\mathcal{T}_\varepsilon$, then fills the missing entries, makes the table $\mathcal{T}_\varepsilon$ closed, and constructs from $\mathcal{T}_\varepsilon$ a new FDA A^ε, resulting in a new conjectured FDFA $\mathcal{F}_1$, shown in Figure 10.

With $\mathcal{F}_1$ at hand, the learner can ask the teacher the equivalence query $\mathrm{EQ}(\mathcal{F}_1)$; she answers "no" with for instance the counterexample (ε, bab). According to Definition 9, (ε, bab) is a negative counterexample, since it is captured by $\mathcal{F}_1$ but clearly $\varepsilon(bab)^\omega \notin L = (ab)^\omega$. The learner has to refine again the progress DFA A^ε: after asking the sequence of membership queries $\mathrm{MQ}(\varepsilon, \langle s_0 = \varepsilon\rangle \cdot bab)$, $\mathrm{MQ}(\varepsilon, \langle s_1 = \varepsilon\rangle \cdot ab)$, $\mathrm{MQ}(\varepsilon, \langle s_2 = a\rangle \cdot b)$, and $\mathrm{MQ}(\varepsilon, \langle s_3 = ab\rangle \cdot \varepsilon)$, he finds the experiment ab distinguishing $\varepsilon \cdot b$ from ε. In general, on receiving a negative counterexample (u, v), the sequence of membership queries has different results for the first query $(\tilde{u}, \varepsilon \cdot v)$ and the last query $(\tilde{u}, A^{\tilde{u}}(v) \cdot \varepsilon)$. This is because $\tilde{u} \cdot (\varepsilon \cdot v)^\omega \notin L$ by assumption while $\tilde{u} \cdot (A^{\tilde{u}}(v) \cdot \varepsilon)^\omega \in L$ since $A^{\tilde{u}}(v)$ is an accepting state. The learner thus uses the experiment ab to update the table $\mathcal{T}_\varepsilon$ as seen before and constructs a new progress DFA A^ε out of $\mathcal{T}_\varepsilon$, which are shown in Figure 11.

The learner is ready to ask the equivalence query $\mathrm{EQ}(\mathcal{F}_2)$ obtaining yet another time "no" as answer, together with a counterexample, say (a, ab), which is again a negative counterexample. Since $\tilde{u} = M(a) = \varepsilon$ and $\varepsilon \cdot (ab)^\omega \in L$, the learner this time has to refine the leading DFA M.

Refinement of the leading DFA M. Assume that the learner has received a negative counterexample (u, v); the case of positive counterexamples is symmetric and thus omitted here. Let $\tilde{u} = M(u)$; by definition we have $uv^\omega \notin L$ while $\tilde{u}v^\omega \in L$. Let $n = |u|$ and for every $i \in [1 \cdots n]$, let $s_i = M(u[1..i])$ be the state



	$(\varepsilon, \varepsilon)$
ε	0
a	0
b	0

$\approx^\varepsilon_P$	ε	b	ab
ε	0	0	1
a	0	1	0
ab	1	0	1
b	0	0	0
aa	0	0	0
aba	0	1	0
abb	0	0	0
ba	0	0	0
bb	0	0	0

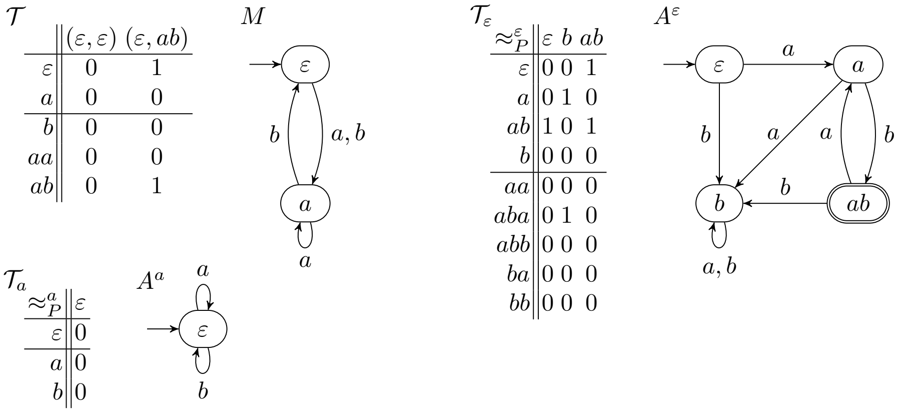
Fig. 11. The intermediate FDFA $\mathcal{F}_2$ and its corresponding tables while learning $(ab)^\omega$

in M after reading the first i letters of u. In particular, $s_0 = \varepsilon$. As in the previous analyses, there is the sequence of membership queries $\mathrm{MQ}(s_0 \cdot u[1..n], v)$, $\mathrm{MQ}(s_1 \cdot u[2..n], v)$, and so on, up to $\mathrm{MQ}(s_n \cdot \varepsilon, v)$. This sequence has different results for the first and the last query since $s_0 \cdot u[1..n] \cdot v^\omega \notin L$ while $s_n \cdot \varepsilon \cdot v^\omega \in L$. Therefore, the learner can find the first experiment $(u[j+1..n], v)$ such that $s_{j-1} \cdot u[j] \cdot u[j+1..n] \cdot v^\omega \notin L$ while $s_j \cdot u[j+1..n] \cdot v^\omega \in L$, which means that the experiment $(u[j+1..n], v)$ can be used to distinguish $s_{j-1} \cdot u[j]$ from s_j.

Consider again the FDFA $\mathcal{F}_2$ shown in Figure 11 and the negative counterexample (a, ab): the learner finds the experiment (ε, ab) to distinguish $\varepsilon \cdot a$ from ε. As usual, after updating the leading table $\mathcal{T}$ by adding the experiment (ε, ab) and closing $\mathcal{T}$, the learner constructs a new conjecture leading DFA M, which is depicted in Figure 12. Moreover, for every new state $u \in U$ of $\mathcal{T}$, he initializes a new progress table $\mathcal{T}_u$ and builds the corresponding progress DFA A^u as before; see for example the progress table $\mathcal{T}_a$ and the progress DFA A^a depicted in Figure 12.

The learner asks the teacher whether $\mathcal{F}_2$ is correct. Assume that the teacher answers "no" with the counterexample (bb, ab) which is negative. By following the same procedure as above, he finds the experiment (b, ab) to distinguish $\varepsilon \cdot b$ from a, which is used to update the leading table $\mathcal{T}$ with experiment (b, ab) and to add the new progress DFA A^b for the state b of M, obtaining the FDFA $\mathcal{F}_4$ shown in Figure 13.

By comparing the leading DFA M in $\mathcal{F}_4$ in Figure 13 with the one in Figure 7, we can see that they are the same, so M is not going to be changed anymore since it is already consistent with the one induced by $\backsim_L$ in Definition 8. However, the progress DFA A^a is still not correct so the teacher answers "no" to the equivalence query $\mathrm{EQ}(\mathcal{F}_4)$ posed by the learner. Assume that the teacher returns the counterexample (a, ba) which is positive. The learner then finds the experiment a to refine the progress DFA A^a and finally he generates the new conjectured FDFA $\mathcal{F}_5$ depicted in Figure 14.



$\mathcal{T}$

	$(\varepsilon,\varepsilon)$	(ε,ab)
ε	0	1
a	0	0
b	0	0
aa	0	0
ab	0	1

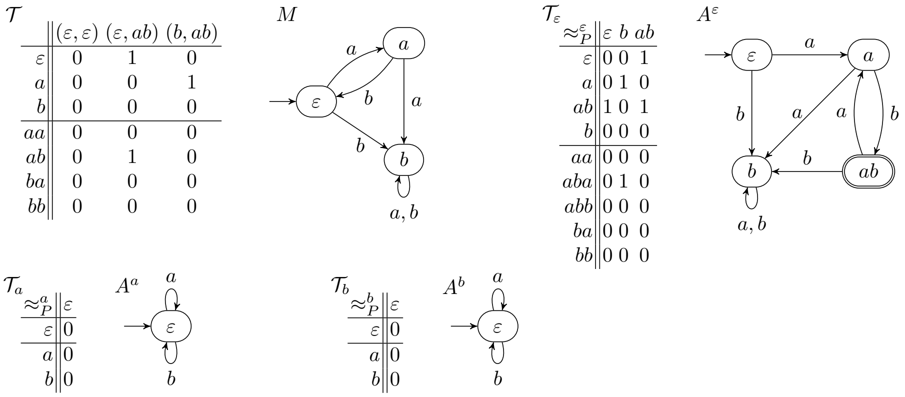
$\mathcal{T}_\varepsilon$

$\approx^\varepsilon_P$	ε	b	ab
ε	0	0	1
a	0	1	0
ab	1	0	1
b	0	0	0
aa	0	0	0
aba	0	1	0
abb	0	0	0
ba	0	0	0
bb	0	0	0

$\mathcal{T}_a$

$\approx^a_P$	ε
ε	0
a	0
b	0

Fig. 12. The intermediate FDFA $\mathcal{F}_3$ and its corresponding tables while learning $(ab)^\omega$



$\mathcal{T}$

	$(\varepsilon,\varepsilon)$	(ε,ab)	(b,ab)
ε	0	1	0
a	0	0	1
b	0	0	0
aa	0	0	0
ab	0	1	0
ba	0	0	0
bb	0	0	0

$\mathcal{T}_\varepsilon$

$\approx^\varepsilon_P$	ε	b	ab
ε	0	0	1
a	0	1	0
ab	1	0	1
b	0	0	0
aa	0	0	0
aba	0	1	0
abb	0	0	0
ba	0	0	0
bb	0	0	0

$\mathcal{T}_a$

$\approx^a_P$	ε
ε	0
a	0
b	0

$\mathcal{T}_b$

$\approx^b_P$	ε
ε	0
a	0
b	0

Fig. 13. The FDFA $\mathcal{F}_4$ and its corresponding tables while learning $(ab)^\omega$

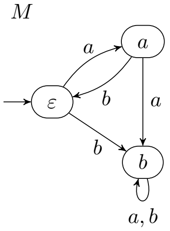
The FDFA $\mathcal{F}_5$ is still not the right conjecture and the teacher answers again "no" to the equivalence query for it. Assume that the returned counterexample is (a, aba) which is clearly negative. As before, the learner refines the progress DFA A^a and gets a new conjecture FDFA $\mathcal{F}_6$ shown in Figure 15.

The teacher now answers "yes" to the equivalence query $\mathrm{EQ}(\mathcal{F}_6)$ and the learner has completed his job.

The FDFA learner. By means of the previous example, we have introduced informally the ω-regular language learning algorithm, which is formalized in Algorithm 2 as the periodic FDFA learner. We can note that the learning procedure we described in the running example follows exactly the steps of the algorithm. In Algorithm 2 we have functions acting on DFAs that are specialized for the leading DFA M (whose with subscript l) and functions specialized

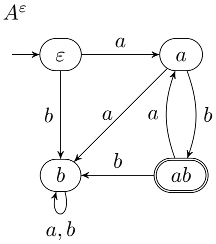
$\mathcal{T}$

	$(\varepsilon,\varepsilon)$	(ε, ab)	(b, ab)
ε	0	1	0
a	0	0	1
b	0	0	0
aa	0	0	0
ab	0	1	0
ba	0	0	0
bb	0	0	0



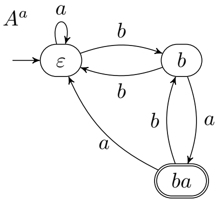
$\mathcal{T}_\varepsilon$

$\approx_P^\varepsilon$	ε	b	ab
ε	0	0	1
a	0	1	0
ab	1	0	1
b	0	0	0
aa	0	0	0
aba	0	1	0
abb	0	0	0
ba	0	0	0
bb	0	0	0



$\mathcal{T}_a$

$\approx_P^a$	ε	a
ε	0	0
b	0	1
ba	1	0
a	0	0
bb	0	0
baa	0	0
bab	0	1



$\mathcal{T}_b$

$\approx_P^b$	ε
ε	0
a	0
b	0

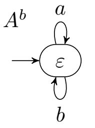


Fig. 14. The intermediate FDFA $\mathcal{F}_5$ and its corresponding tables while learning $(ab)^\omega$

for the progress DFAs (whose with subscript p). Since for the refinement of the progress DFA $A^{\tilde{u}}$ the learner does not need the word u but just $\tilde{u}$, function $\mathit{FindDistinguishingExperiment}_p$ takes the parameter instance $\tilde{u}$ instead of u.

The soundness and completeness of Algorithm 2 are guaranteed by Theorem 5.

Theorem 5. *Assume that L is the target ω-regular language. Algorithm 2 terminates and returns a periodic FDFA $\mathcal{F}$ capturing the set of decompositions $\{\, (u, v) \in \Sigma^* \times \Sigma^+ \mid uv^\omega \in L \,\}$.*

Clearly, the FDFA $\mathcal{F}$ returned by Algorithm 2 is a correct conjecture because the teacher has approved it; Algorithm 2 terminates because: 1) we can discover new states, i.e., new equivalence classes in $\Sigma^*/_{\backsim_L}$ or in $\Sigma^*/_{\approx_P^{\tilde{u}}}$, whenever receiving a counterexample (u, v) from the teacher, where $\tilde{u} = M(u)$ (cf. [72]); and 2) $\backsim_L$ and $\approx_P^{\tilde{u}}$ for any $u \in \Sigma^*$ are all right congruences of finite index (cf. [11]).

The complexity of Algorithm 2 is stated in Theorem 6; let the length of a decomposition (u, v) be the sum of the lengths of u and v, i.e., $|(u, v)| = |u| + |v|$.

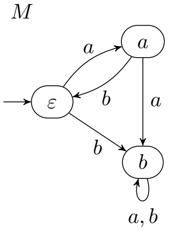
Theorem 6. *Given a target ω-regular language L, let n be the sum of the indexes of the right congruences, i.e., $n = |\backsim_L| + \sum_{[u]_{\backsim_L} \in \Sigma^*/_{\backsim_L}} |\approx_P^u|$, and m be the maximum length of any counterexample (u, v) returned by the teacher.*

1. *Algorithm 2 terminates on receiving at most n counterexamples.*
2. *The number of membership queries is in $\mathcal{O}(n^2 \cdot |\Sigma| + n \cdot m)$.*

The reason why Algorithm 2 terminates on receiving at most n counterexamples is obvious since there are n states in the periodic FDFA of L. The reason why

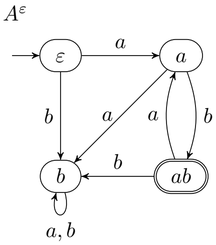
$\mathcal{T}$

	$(\varepsilon, \varepsilon)$	(ε, ab)	(b, ab)
ε	0	1	0
a	0	0	1
b	0	0	0
aa	0	0	0
ab	0	1	0
ba	0	0	0
bb	0	0	0



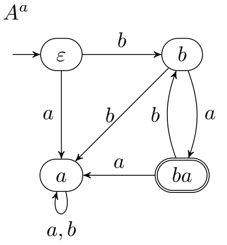
$\mathcal{T}_\varepsilon$

$\approx_P^\varepsilon$	ε	b	ab
ε	0	0	1
a	0	1	0
ab	1	0	1
b	0	0	0
aa	0	0	0
aba	0	1	0
abb	0	0	0
ba	0	0	0
bb	0	0	0



$\mathcal{T}_a$

$\approx_P^a$	ε	a	ba
ε	0	0	1
b	0	1	0
ba	1	0	1
a	0	0	0
bb	0	0	0
baa	0	0	0
bab	0	1	0
aa	0	0	0
ab	0	0	0



$\mathcal{T}_b$

$\approx_P^b$	ε
ε	0
a	0
b	0

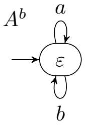


Fig. 15. The final FDFA $\mathcal{F}_6$ and its corresponding tables while learning $(ab)^\omega$

the number of membership queries is in $\mathcal{O}(n^2 \cdot |\Sigma| + n \cdot m)$ is the following: let l be the number of states in the leading DFA M and $p_1, p_2, \ldots, p_l$ be the number of states in the progress DFAs, respectively; we have $l + \sum_{i=1}^{l} p_i = n$. By Theorem 4, a DFA with k_1 states and longest counterexample of length k_2 can be learned by at most $k_1^2 \cdot |\Sigma| + k_1 \cdot k_2$ membership queries, thus we need at most $l^2 \cdot |\Sigma| + l \cdot m + \sum_{i=1}(p_i^2 \cdot |\Sigma| + p_i \cdot m) \in \mathcal{O}(n^2 \cdot |\Sigma| + n \cdot m)$ membership queries.

5.4 Learning Büchi Automata

In the previous section we presented the FDFA learning algorithm, which is our secret ingredient in learning a BA: we first learn an FDFA $\mathcal{F}$ and then transform the learned $\mathcal{F}$ to a BA. This is just one sentence introduction to the BA learning algorithm; there are however several details than need to be concretized in order to get a working algorithm.

Overview of the BA learning framework. In the following we begin with an introduction of the framework presented in [73] for learning BA as depicted in Figure 16. In this section, we let the ω-regular language L be the target language and we assume that we already have a BA teacher who knows the language L and can answer membership and equivalence queries about L. In order to distinguish membership and equivalence queries posed by the FDFA learner and the BA learner, we use a superscript like FDFA and BA to mark

Algorithm 2: The Periodic FDFA Learner

```
Initialize leading table $\mathcal{T} = (U, V, T)$ with $U = \{\varepsilon\}$ and $V = \{(\varepsilon, \varepsilon)\}$;
$\mathit{CloseTable}_l(\mathcal{T}, \mathrm{MQ}(\,\cdot\,))$ and let $M = \mathit{Aut}_l(\mathcal{T})$;
foreach $u \in U$ do
    Initialize progress table $\mathcal{T}_u = (U_u, V_u, T_u)$ with $U_u = \{\varepsilon\}$ and $V_u = \{\varepsilon\}$;
    $\mathit{CloseTable}_p(\mathcal{T}_u, \mathrm{MQ}(\,\cdot\,))$ and let $A^u = \mathit{Aut}_p(\mathcal{T}_u)$;
Let $(a, (u, v))$ be the teacher's response on $\mathrm{EQ}(\mathcal{F})$;
while $a =$ "no" do
    Let $\tilde{u} = M(u)$;
    if $\mathrm{MQ}(\tilde{u}, v) \neq \mathrm{MQ}(u, v)$ then
        $V = V \cup \mathit{FindDistinguishingExperiment}_l(u, v)$;
        $\mathit{CloseTable}_l(\mathcal{T}, \mathrm{MQ}(\,\cdot\,))$ and let $M = \mathit{Aut}_l(\mathcal{T})$;
        foreach newly added $u \in U$ do
            Initialize progress table $\mathcal{T}_u = (U_u, V_u, T_u)$ with $U_u = \{\varepsilon\}$ and
             $V_u = \{\varepsilon\}$;
            $\mathit{CloseTable}_p(\mathcal{T}_u, \mathrm{MQ}(\,\cdot\,))$ and let $A^u = \mathit{Aut}_p(\mathcal{T}_u)$;
    else
        $V_{\tilde{u}} = V_{\tilde{u}} \cup \mathit{FindDistinguishingExperiment}_p(\tilde{u}, v)$;
        $\mathit{CloseTable}_p(\mathcal{T}_{\tilde{u}}, \mathrm{MQ}(\,\cdot\,))$ and let $A^{\tilde{u}} = \mathit{Aut}_p(\mathcal{T}_{\tilde{u}})$;
    Let $(a, (u, v))$ be the teacher's response on $\mathrm{EQ}(\mathcal{F})$;
return $\mathcal{F}$;
```

queries from the FDFA learner and the BA learner, respectively. For instance, the membership query $\mathrm{MQ}^{\mathrm{FDFA}}(\,\cdot\,)$ is posed by the FDFA learner while $\mathrm{MQ}^{\mathrm{BA}}(\,\cdot\,)$ is asked by the BA learner.

The BA learner, shown in Figure 16 surrounded by the dashed box, has three components, namely the FDFA learner, the component transforming an FDFA $\mathcal{F}$ to a BA $B_{\mathcal{F}}$, and the counterexample analysis component. The BA learner first uses the FDFA learner to learn an FDFA $\mathcal{F}$ by means of membership and equivalence queries. This makes some problem the BA learner has to solve: on the one side, in order to answer queries posed by the FDFA learner, the BA learner needs an FDFA teacher to answer membership and equivalence queries about the target periodic FDFA of L; one the other side, there is only a BA teacher who can answer queries about the target language.

In this situation, the BA learner acts as an interface between the FDFA teacher and the FDFA learner and tries to pretend to be an FDFA teacher when he has to answer the queries from the FDFA learner. In other words, the BA learner becomes the FDFA teacher by interacting with the BA teacher. To that end, the FDFA teacher answers to a membership query $\mathrm{MQ}^{\mathrm{FDFA}}(u, v)$ by simply forwarding the answer to the membership query $\mathrm{MQ}^{\mathrm{BA}}(uv^{\omega})$ obtained from the BA teacher to the FDFA learner, which is trivial.

It is, however, more tricky for the FDFA teacher to answer an equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F})$ posed by the FDFA learner. The FDFA teacher first needs to transform the conjectured FDFA $\mathcal{F}$ to a BA $B_{\mathcal{F}}$ and then poses the equivalence query $\mathrm{EQ}^{\mathrm{BA}}(B_{\mathcal{F}})$ to the BA teacher. If the BA teacher answers "yes", the BA

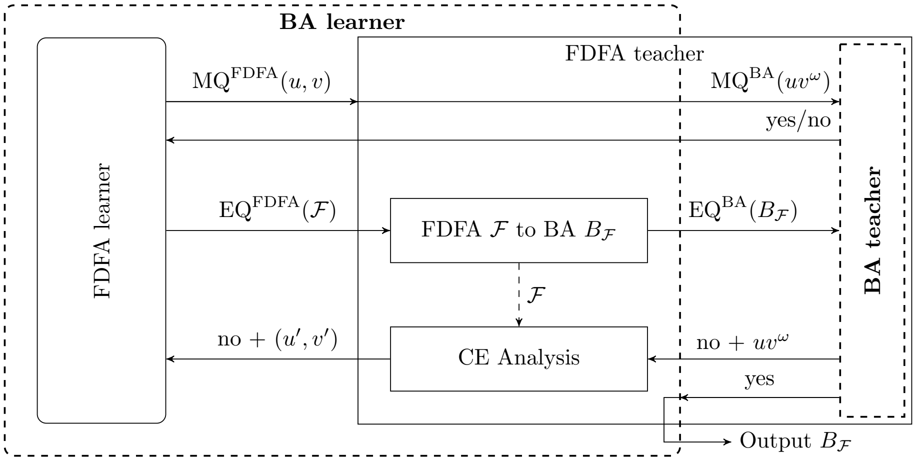

BA learner
FDFA teacher
FDFA learner
BA teacher
$\mathrm{MQ}^{\mathrm{FDFA}}(u, v)$
$\mathrm{MQ}^{\mathrm{BA}}(uv^{\omega})$
yes/no
$\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F})$
FDFA $\mathcal{F}$ to BA $B_{\mathcal{F}}$
$\mathrm{EQ}^{\mathrm{BA}}(B_{\mathcal{F}})$
$\mathcal{F}$
no + (u', v')
CE Analysis
no + uv^{ω}
yes
Output $B_{\mathcal{F}}$

Fig. 16. Overview of the BA learning framework based on FDFA learning

learner first receives the answer and then outputs the BA $B_{\mathcal{F}}$ as he has completed the learning task. Otherwise the BA teacher returns "no" together with a counterexample uv^{ω} given as a decomposition (u, v). The BA learner then performs the counterexample analysis and, by acting as an FDFA teacher, he feeds the FDFA learner with a valid decomposition (u', v') which satisfies Definition 9, so that the FDFA learner can further refine the current FDFA $\mathcal{F}$.

Note that in Figure 16 there is a dashed arrow labeled with $\mathcal{F}$ entering the counterexample analysis block: it indicates the fact that the FDFA teacher needs to use the current conjectured FDFA $\mathcal{F}$ in the analysis of the counterexample, as we will see later. We want to remark that, according to Figure 16, the BA teacher is oblivious of the FDFA learner, since she only sees a BA learner interacting with her and similarly, the FDFA learner does not know that there is a BA teacher since it is the FDFA teacher that is answering his queries.

From the framework depicted in Figure 16, we get the rough idea about how to build a BA learner out of an FDFA learner. Yet there are still few details we have to sort out:

- How can we transform an FDFA $\mathcal{F}$ to a BA $B_{\mathcal{F}}$?
- How can we get a valid counterexample (u', v') for the FDFA learner out of a counterexample (u, v) returned by the BA teacher?

The answers to the above questions are the missing bricks we need to build a BA learner based on an FDFA learner. In the following, we first answer the question on how to do the transformation from an FDFA to a BA and then introduce the counterexample analysis through an example.

From FDFA $\mathcal{F}$ to BA $B_{\mathcal{F}}$. Assume that we want to learn a BA which accepts the ω-regular language $L = (ab)^{\omega}$ over $\Sigma = \{a, b\}$. To that end, the BA learner

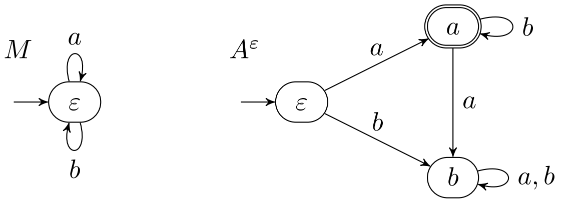


Fig. 17. An FDFA $\mathcal{F}$ such that $\mathrm{UP}(\mathcal{F})$ does not characterize an ω-regular language

first initializes an FDFA learner which constructs the initial conjectured FDFA $\mathcal{F}_0$ as depicted in Figure 9 via membership queries. On receiving the conjectured FDFA $\mathcal{F}_0$, the BA learner has to construct a BA $B_{\mathcal{F}_0}$ from $\mathcal{F}_0$ which we illustrate in the following.

To answer an equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F})$, the BA learner needs fist to covert $\mathcal{F}$ into a BA $B_{\mathcal{F}}$ in order to exploit the BA teacher to answer the query. The first question one may ask in doing this is:

- Is it possible to construct a *precise* BA $B_{\mathcal{F}}$ for each given FDFA $\mathcal{F}$ such that $\mathrm{UP}(\mathcal{L}(B_{\mathcal{F}})) = \mathrm{UP}(\mathcal{F})$?

The answer is actually *no*, as the following example shows.

Example 3 (Non-regular ω-language accepted by an FDFA [73]). Consider the FDFA $\mathcal{F}$ depicted in Figure 17 where $\mathrm{UP}(\mathcal{F}) = \bigcup_{n=0}^{\infty}\{a,b\}^* \cdot (ab^n)^\omega$. Assume that $\mathrm{UP}(\mathcal{F})$ characterizes an ω-regular language L. It is claimed in [11] that for every ω-regular language, there exists a periodic FDFA recognizing it and the index of each right congruence of the periodic FDFA is finite. Therefore we let $\mathcal{F}'$ be the periodic FDFA of L and we know that the right congruence $\approx_P^\varepsilon$ of $\mathcal{F}'$ is of finite index. However, we can show that the right congruence $\approx_P^\varepsilon$ of $\mathcal{F}'$ has to be of infinite index. Observe that $ab^k \not\approx_P^\varepsilon ab^j$ for any $k, j \geq 1$ and $k \neq j$, since $\varepsilon \cdot (ab^k \cdot ab^k)^\omega \in \mathrm{UP}(\mathcal{F})$ and $\varepsilon \cdot (ab^j \cdot ab^k)^\omega \notin \mathrm{UP}(\mathcal{F})$ according to Definition 8. It follows that $\approx_P^\varepsilon$ is of infinite index. Contradiction. Thus we conclude that $\mathrm{UP}(\mathcal{F})$ cannot characterize an ω-regular language.

Therefore, in general, one can not construct a BA $B_{\mathcal{F}}$ from an FDFA $\mathcal{F}$ such that $\mathrm{UP}(\mathcal{L}(B_{\mathcal{F}})) = \mathrm{UP}(\mathcal{F})$. The authors of [73] suggested two BA constructions to approximate the set of ultimately periodic words $\mathrm{UP}(\mathcal{F})$: the under-approximation and the over-approximation construction. In this work, we only introduce the *under-approximation* construction from [73], which produces a BA $B_{\mathcal{F}}$ that under-approximates $\mathrm{UP}(\mathcal{F})$, i.e., $\mathrm{UP}(\mathcal{L}(B_{\mathcal{F}})) \subseteq \mathrm{UP}(\mathcal{F})$. This construction was originally proposed by Calbrix *et al.* in [26].

We first give the main idea behind the under-approximation method and then give its formal definition. Let $\mathcal{F}$ be the FDFA $\mathcal{F} = (M, \{A^u\})$ with $M = (Q, \bar{q}, \delta, \emptyset)$ and $A^u = (Q_u, s_u, \delta_u, F_u)$ for each $u \in Q$. Let $M_v^s = (Q, s, \delta, \{v\})$ and $(A^u)_v^s = (Q_u, s, \delta_u, \{v\})$ be the DFAs obtained from M and A^u by setting their initial state and accepting states to s and $\{v\}$, respectively. We define

$N_{(u,v)} = \{\, v^\omega \mid M(uv) = M(u) \wedge v \in \mathcal{L}((A^u)^{s_u}_v)\,\}$, which contains only the words $v \in \mathcal{L}((A^u)^{s_u}_v)$ such that $u = M(u) = M(uv)$. Recall that we use words u and v to represent the states in the DFAs. Therefore, according to the acceptance condition of FDFAs in Definition 6, we have that $\mathrm{UP}(\mathcal{F}) = \bigcup_{u \in Q, v \in F_u} \mathcal{L}(M^{\bar{q}}_u) \cdot N_{(u,v)}$ where $\mathcal{L}(M^{\bar{q}}_u)$ contains the set of finite prefixes and $N_{(u,v)}$ contains the set of finite periodic words for every state pair (u, v).

We construct $B_\mathcal{F}$ by approximating the set $N_{(u,v)}$, i.e., the set of finite periodic words. We first define the FA $P_{(u,v)} = (Q_{(u,v)}, s_{(u,v)}, \delta_{(u,v)}, \{f_{(u,v)}\}) = M^u_u \times (A^u)^{s_u}_v \times (A^u)^v_v$ and let $\underline{N}_{(u,v)} = \mathcal{L}(P_{(u,v)})^\omega$. Recall that the notation $\times$ here is the intersection operation of FAs. Then we can construct the BA $(Q_{(u,v)} \cup \{f\}, s_{(u,v)}, \delta_{(u,v)} \cup \delta_f, \{f\})$ recognizing $\underline{N}_{(u,v)}$ where f is a "fresh" state and $\delta_f = \{(f, \varepsilon, s_{(u,v)}), (f_{(u,v)}, \varepsilon, f)\}$. Note that ε transitions can be taken without consuming any letters and can be removed by standard methods in automata theory, see, e.g., [56]. Intuitively, we under-approximate the set $N_{(u,v)}$ as $\underline{N}_{(u,v)}$ by only keeping $v^\omega \in N_{(u,v)}$ if $A^u(v) = A^u(v \cdot v)$ where $v \in \Sigma^+$.

In Definition 10 we provide the construction procedure for a BA $B_\mathcal{F}$ such that $\mathrm{UP}(\mathcal{L}(B_\mathcal{F})) = \bigcup_{u \in Q, v \in F_u} \mathcal{L}(M^{\bar{q}}_u) \cdot \underline{N}_{(u,v)} = \bigcup_{u \in Q, v \in F_u} \mathcal{L}(M^{\bar{q}}_u) \cdot (\mathcal{L}(P_{(u,v)}))^\omega$, as originally proposed in [26].

Definition 10 ([73]). *Let $\mathcal{F} = (M, \{A^u\})$ be an FDFA where $M = (Q, \bar{q}, \delta, \emptyset)$ and $A^u = (Q_u, s_u, \delta_u, F_u)$ for each $u \in Q$. Let $(Q_{(u,v)}, s_{(u,v)}, \delta_{(u,v)}, \{f_{(u,v)}\})$ be a BA recognizing $\underline{N}_{(u,v)}$. Then the BA $B_\mathcal{F}$ is defined as the tuple $B_\mathcal{F} = (Q_{B_\mathcal{F}}, \bar{q}_{B_\mathcal{F}}, \delta_{B_\mathcal{F}}, F_{B_\mathcal{F}})$ where*

- $Q_{B_\mathcal{F}} = Q \cup \bigcup\limits_{u \in Q, v \in F_u} Q_{(u,v)}$,
- $\bar{q}_{B_\mathcal{F}} = \bar{q}$,
- $\delta_{B_\mathcal{F}} = \delta \cup \bigcup\limits_{u \in Q, v \in F_u} \delta_{(u,v)} \cup \bigcup\limits_{u \in Q, v \in F_u} \{(u, \varepsilon, s_{(u,v)})\}$, *and*
- $F_{B_\mathcal{F}} = \bigcup\limits_{u \in Q, v \in F_u} \{f_{(u,v)}\}$

Intuitively, we connect the leading DFA M to the BA recognizing $\underline{N}_{(u,v)}$ by linking the state u of M and the initial state $s_{(u,v)}$ of the BA with an ε-transition for every state pair (u, v) where $v \in F_u$.

We now present Lemma 2 which is used later for the counterexample analysis.

Lemma 2 (cf. [73, Lemma 4]). *Let $\mathcal{F}$ be an FDFA, and $B_\mathcal{F}$ be the BA constructed from $\mathcal{F}$ according to Definition 10. If (u, v^k) is accepted by $\mathcal{F}$ for every $k \geq 1$, then $uv^\omega \in UP(\mathcal{L}(B_\mathcal{F}))$.*

The following theorem is the main result of our BA construction. We refer the interested reader to [72] for the proofs of Lemma 2 and Theorem 7.

Theorem 7 (cf. [73, Lemma 3]). *Let $\mathcal{F}$ be the current conjectured FDFA and $B_\mathcal{F}$ be the BA constructed from $\mathcal{F}$ according to Definition 10. Let n and k be the number of states in the leading DFA and the largest progress DFA of $\mathcal{F}$, respectively. Then*

- *the number of states in $B_\mathcal{F}$ is in $\mathcal{O}(n^2k^3)$;*

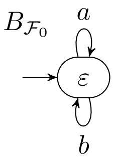


Fig. 18. The BA $B_{\mathcal{F}_0}$ constructed for answering the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_0)$, with $\mathcal{F}_0$ shown in Figure 9

- $UP(\mathcal{L}(B_{\mathcal{F}})) \subseteq UP(\mathcal{F})$;
- $UP(\mathcal{L}(B_{\mathcal{F}})) = UP(\mathcal{F})$ *if* $\mathcal{F}$ *is the periodic FDFA accepting* $UP(\mathcal{F})$.

For instance, the initial BA $B_{\mathcal{F}_0}$ constructed from the FDFA $\mathcal{F}_0$ shown in Figure 9 is depicted in Figure 18. The state space of $Q_{(u,v)}$ of $B_{\mathcal{F}_0}$ defined in Definition 10 is empty since F_ε of A^ε in $\mathcal{F}_0$ is empty.

The BA $B_{\mathcal{F}_0}$ is clearly not a right conjecture so the BA teacher answers "no" for the equivalence query $\mathrm{EQ}^{\mathrm{BA}}(B_{\mathcal{F}_0})$ together with a counterexample, say $(ab)^\omega \in \mathcal{L}(B_{\mathcal{F}_0}) \ominus L$, given by the decomposition (ε, ab). Since the counterexample (ε, ab) from the BA teacher is a positive counterexample for the FDFA learner, according to Definition 9, the FDFA teacher who is disguised by the BA learner just sets (u', v') to be (ε, ab) in the counterexample analysis and returns it to the FDFA learner as counterexample for the "no" answer. We remark that if the BA learner applies the counterexample analysis to the valid positive counterexample (ε, ab) for the FDFA learner, the procedure also outputs a valid positive counterexample which satisfies Definition 9. In other words, in practice the counterexample analysis on the received counterexample directly generates a valid counterexample, so the BA learner does not have to decide whether the counterexample received from the BA teacher is valid for the FDFA learner.

Since he has received a negative answer with a counterexample for the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_0)$, the FDFA learner refines the current FDFA $\mathcal{F}_0$ according to the received counterexample, as we have seen in Section 5.3, and then poses the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_1)$ for the new conjectured FDFA $\mathcal{F}_1$, which is shown in Figure 10.

The BA learner then builds from $\mathcal{F}_1$ the under-approximation BA $B_{\mathcal{F}_1}$, which is depicted in Figure 19: observe that the ω-word $(bab)^\omega \in \mathcal{L}(B_{\mathcal{F}_1})$ is accepted by $\mathcal{F}_1$ since the decomposition (ε, bab) is accepted by $\mathcal{F}_1$; the BA $\underline{N}_{(\varepsilon,ab)}$ is defined as the DFA $M^\varepsilon_\varepsilon \times (A^\varepsilon)^\varepsilon_{ab} \times (A^\varepsilon)^{ab}_{ab}$ augmented with an extra state f and it is shown in the dashed box in Figure 19.

Again, the conjectured BA $B_{\mathcal{F}_1}$ is not the right conjecture. The BA teacher answers the equivalence query for the BA $B_{\mathcal{F}_1}$ with "no" and, say, the counterexample $(bab)^\omega \in \mathcal{L}(B_1) \ominus L$, given as the decomposition (b, abb).

The counterexample (b, abb) is however not a valid counterexample for the FDFA learner according to Definition 9 since $(bab)^\omega \notin \mathrm{UP}(L)$ and (b, abb) is not captured by the current FDFA $\mathcal{F}_1$. Suppose that the BA learner feeds the FDFA learner with the counterexample (b, abb); it is easy to verify that he is not able to

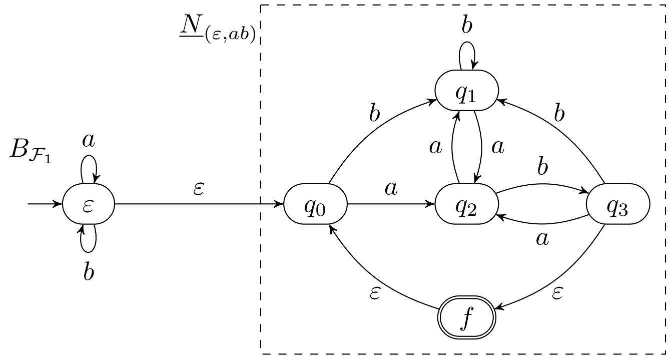


Fig. 19. The BA $B_{\mathcal{F}_1}$ constructed for answering the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_1)$, with $\mathcal{F}_1$ shown in Figure 10

identify new states with the help of (b, abb), so he is going to conjecture again the FDFA $\mathcal{F}_1$. Therefore, if the FDFA learner repeatedly poses the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_1)$ for $\mathcal{F}_1$ and the BA teacher always answers (b, abb), the learning procedure is going to get stuck in an infinite loop. This motivates the need of the counterexample analysis, which ensures that the counterexample returned by the BA teacher can be adapted to a valid counterexample for the FDFA learner which allows him to refine the conjectured FDFA.

Counterexample analysis for the FDFA teacher. In order to ensure the termination of the learning procedure, the BA learner has to execute the counterexample analysis so to get a valid counterexample for the FDFA learner out of (b, abb).

To distinguish the different counterexamples from the different teachers, we define the counterexample from the BA teacher as follows.

Definition 11 (Counterexample for the FDFA teacher). *Let L be the target ω-regular language and $B_{\mathcal{F}}$ be the current conjectured BA. We say a counterexample (u, v) with $uv^\omega \in \mathcal{L}(B_{\mathcal{F}}) \ominus L$ is*

- *positive if $uv^\omega \in L$ and $uv^\omega \notin \mathcal{L}(B_{\mathcal{F}})$, and*
- *negative if $uv^\omega \notin L$ and $uv^\omega \in \mathcal{L}(B_{\mathcal{F}})$.*

This is a symmetric definition when compared with the counterexample for the FDFA learner given in Definition 9.

We call a positive counterexample uv^ω *spurious* if $uv^\omega \in \mathrm{UP}(\mathcal{F})$. A spurious positive counterexample (u, v) witnesses that $\mathrm{UP}(\mathcal{L}(B_{\mathcal{F}})) \subset \mathrm{UP}(\mathcal{F})$ holds; the reason for this is that: according to Theorem 7, we have $\mathrm{UP}(\mathcal{L}(B_{\mathcal{F}})) \subseteq \mathrm{UP}(\mathcal{F})$; by the definition of positive counterexample for the FDFA teacher, we have $uv^\omega \notin \mathrm{UP}(\mathcal{L}(B_{\mathcal{F}}))$ yet $uv^\omega \in \mathrm{UP}(\mathcal{F})$ holds.

In order to analyze the counterexample (u, v), it is useful to know how the received counterexample relates with the conjectured FDFA $\mathcal{F}$. For instance, it

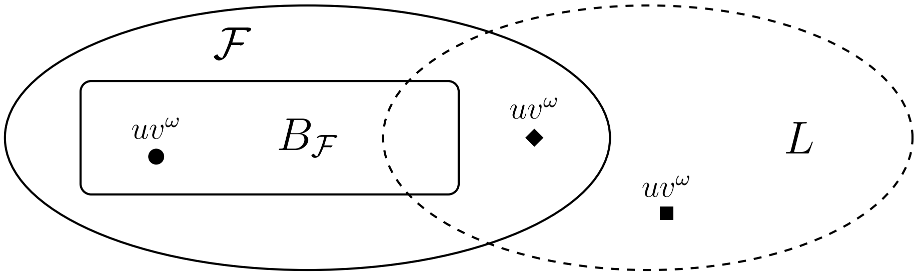


Fig. 20. The cases for the counterexample analysis

may be that $\mathcal{F}$ captures (u, v) but uv^ω is not in the target language, so (u, v) should be rejected; symmetrically, we can have that $\mathcal{F}$ rejects any decomposition of uv^ω but uv^ω is in the target language, so at least (u, v) should be captured; moreover, it may be that $\mathcal{F}$ rejects just (u, v) while capturing another decomposition (u', v') of uv^ω, but $B_\mathcal{F}$ does not accept (u', v'). To distinguish the above cases, which are shown in Figure 20, we use three DFAs accepting different decompositions of uv^ω.

Let $\mathcal{F} = (M, \{A^u\})$ be the current conjectured FDFA. In order to analyze the counterexample (u, v) for the FDFA teacher, we define the following three DFAs, where \$ is a letter not in Σ.

- a DFA $D_\$$ with $\mathcal{L}(D_\$) = \{\, u'\$v' \mid u' \in \Sigma^*, v' \in \Sigma^+, u'v'^\omega = uv^\omega \,\}$,
- a DFA $\mathcal{D}_1$ with $\mathcal{L}(\mathcal{D}_1) = \{\, u'\$v' \mid u' \in \Sigma^*, v' \in \Sigma^*, v' \in \mathcal{L}(A^{M(u')}) \,\}$, and
- a DFA $\mathcal{D}_2$ with $\mathcal{L}(\mathcal{D}_2) = \{\, u'\$v' \mid u' \in \Sigma^*, v' \in \Sigma^*, v' \notin \mathcal{L}(A^{M(u')}) \,\}$.

Intuitively, $D_\$$ accepts every possible decomposition (u', v') of uv^ω; $\mathcal{D}_1$ recognizes every decomposition (u', v') which is captured by $\mathcal{F}$; and $\mathcal{D}_2$ accepts every decomposition (u', v') which is not captured by $\mathcal{F}$.

The DFA $D_\$$ can be constructed according to the procedure presented in [26, 72]; the DFA $\mathcal{D}_1$ can be obtained from $\mathcal{F}$ by simply connecting each state u of M to the initial state s_u of A^u via letter \$; finally, the construction of the DFA $\mathcal{D}_2$ is similar to the one of the DFA $\mathcal{D}_1$ except that we use the complement FDFA $\mathcal{F}^\mathcal{C}$ of $\mathcal{F}$ instead of $\mathcal{F}$. Note that the DFAs $\mathcal{D}_1$ and $\mathcal{D}_2$ in this section are specialized for the periodic FDFA which are different from those defined for all three kinds of FDFAs in [73]. We refer the interested reader to [73] for more details on the different constructions of $\mathcal{D}_1$ and $\mathcal{D}_2$ for the recurrent and syntactic FDFAs.

The analysis for the returned counterexample has first to identify the kind of counterexample it is analyzing and then to deal with it accordingly. More concretely, the three kinds of counterexamples, shown in Figure 20 by means of different shapes, are the following:

case U1: $uv^\omega \in \mathbf{UP}(L)$ while $uv^\omega \notin \mathbf{UP}(\mathcal{F})$ (square). The ω-word uv^ω is a positive counterexample for the FDFA teacher since $uv^\omega \notin \mathrm{UP}(\mathcal{L}(B_\mathcal{F}))$. On the one hand, $uv^\omega \in \mathrm{UP}(L)$ holds and all decompositions of uv^ω should be accepted by $\mathcal{F}$; on the other hand, $uv^\omega \notin \mathrm{UP}(\mathcal{F})$ holds, which indicates that no decomposition of uv^ω is accepted by $\mathcal{F}$. Therefore, in order to further refine $\mathcal{F}$, the FDFA learner needs to make $\mathcal{F}$ accept at least one decomposition (u', v') of uv^ω. That is, the analysis has to find a valid positive counterexample (u', v') out of uv^ω for the FDFA learner such that $v' \notin \mathcal{L}(A^{M(u')})$. This can be easily done by taking a word $u'\$v' \in \mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_2)$. This follows from the fact that DFA $D_\$$ accepts every decomposition of uv^ω and $\mathcal{D}_2$ accepts all decompositions which are not accepted by $\mathcal{F}$. Therefore, a word $u'\$v' \in \mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_2)$ is a valid positive counterexample for the FDFA learner. Note that the analysis tries to find a decomposition (u', v') which is not captured by $\mathcal{F}$ instead of a decomposition not accepted by $\mathcal{F}$. The reason is that a decomposition (u', v') not captured by $\mathcal{F}$ is also a decomposition which is not accepted by $\mathcal{F}$ according to Definition 6. We refer interested reader to [72, Appendix D.3] for more details on case U1.

case U2: $uv^\omega \notin \mathbf{UP}(L)$ while $uv^\omega \in \mathbf{UP}(\mathcal{F})$ (circle). The ω-word uv^ω is a negative counterexample for the FDFA teacher since $uv^\omega \in \mathrm{UP}(\mathcal{L}(B_{\mathcal{F}}))$. It follows that $\mathcal{F}$ should reject every decomposition of uv^ω since $uv^\omega \notin \mathrm{UP}(L)$. In other words, in order to further refine $\mathcal{F}$, the FDFA learner needs to make $\mathcal{F}$ not capture at least one decomposition (u', v') of uv^ω. Therefore, the analysis needs to find a valid negative counterexample (u', v') out of uv^ω for the FDFA learner that is accepted by $\mathcal{F}$. This can be done by taking a word $u'\$v' \in \mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_1)$.

case U3: $uv^\omega \in \mathbf{UP}(L)$ and $uv^\omega \in \mathbf{UP}(\mathcal{F})$ (diamond). The ω-word uv^ω is a spurious positive counterexample since $uv^\omega \in \mathrm{UP}(\mathcal{F})$ but $uv^\omega \notin \mathrm{UP}(\mathcal{L}(B_{\mathcal{F}}))$. On the one hand, this case is quite similar to case U1 since $uv^\omega \in \mathrm{UP}(L)$ and normally we should make $\mathcal{F}$ accept uv^ω; on the other hand, however, differently from U1, $\mathcal{F}$ already accepts at least one decomposition of uv^ω in this case. Suppose that the decomposition (x, y) of uv^ω is accepted by $\mathcal{F}$; according to Lemma 2, there must exist some $k \geq 1$ such that (x, y^k) is not accepted by $\mathcal{F}$ since otherwise we would have $uv^\omega \in \mathcal{L}(B)$. Therefore, similar to case U1, it is possible for the analysis to find a valid positive counterexample (u', v') out of uv^ω for the FDFA learner such that $v' \notin \mathcal{L}(A^{M(u')})$. The word $u'\$v'$ can also be taken from $\mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_2)$. We refer interested reader to [72, Appendix D.3] for more details on case U3.

We remark that from an implementation point of view, checking whether $uv^\omega \in \mathrm{UP}(L)$ can be replaced by a membership query $\mathrm{MQ}^{\mathrm{BA}}(u, v)$ while testing whether $uv^\omega \in \mathrm{UP}(\mathcal{F})$ reduces to checking whether $u\$v \in \mathcal{L}(\mathcal{D}_1)$.

Consider again the conjectured FDFA $\mathcal{F}_1$ and the counterexample (b, abb) returned by the BA teacher, which is a negative counterexample since $b \cdot (abb)^\omega \notin L$. The counterexample (b, abb) falls into case U2 since $b \cdot (abb)^\omega \notin \mathrm{UP}(L)$ while $b \cdot (abb)^\omega \in \mathrm{UP}(\mathcal{F}_1)$. In order to analyze it, the BA learner needs to construct the DFAs $D_\$$ and $\mathcal{D}_1$, which are depicted in Figure 21. For completeness of presentation, Figure 21 shows also the DFA $\mathcal{D}_2$, so to allow the reader to compare it with $\mathcal{D}_1$. Assume that the BA learner gets the word $\varepsilon\$bab \in \mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_1)$, which gives a negative counterexample (ε, bab) for the FDFA learner.

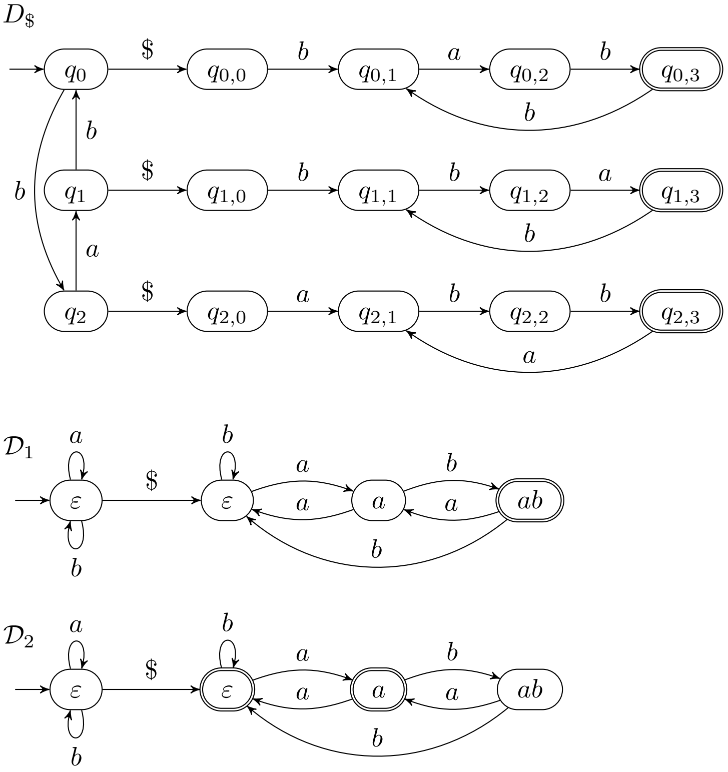


Fig. 21. The three DFAs $D_{\$}$, $\mathcal{D}_1$, and $\mathcal{D}_2$ for the FDFA $\mathcal{F}_1$ from Figure 10 and the counterexample (b, abb)

As seen in Section 5.3, with the negative counterexample (ε, bab) at hand, the FDFA learner is able to get the refined conjecture $\mathcal{F}_2$ depicted in Figure 11. To answer the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_2)$, the BA learner has again to construct a BA $B_{\mathcal{F}_2}$, depicted in Figure 22, from $\mathcal{F}_2$; then he poses the equivalence query $\mathrm{EQ}^{\mathrm{BA}}(B_{\mathcal{F}_2})$ to the BA teacher. Note that we have $\mathrm{UP}(\mathcal{F}_2) = \mathrm{UP}(\mathcal{L}(B_{\mathcal{F}_2})) = \{a, b\}^*(ab)^\omega$ for the constructed BA $B_{\mathcal{F}_2}$. This follows from the fact that $\mathcal{F}_2$ is a periodic FDFA accepting $\{a, b\}^*(ab)^\omega$, which results in $\mathrm{UP}(\mathcal{F}_2) = \mathrm{UP}(\mathcal{L}(B_{\mathcal{F}_2}))$ according to Theorem 7.

The BA teacher answers again "no" with a counterexample, say (a, ab), which is already a negative counterexample for the FDFA learner. However, the BA learner is not aware of this fact, thus he performs a counterexample analysis on (a, ab). In order to analyze the counterexample $a \cdot (ab)^\omega$, the BA learner first builds the two DFAs $D_{\$}$ and $\mathcal{D}_1$ shown in Figure 23.

Suppose that this time the BA learner takes the word $a\$ab \in \mathcal{L}(D_{\$}) \cap \mathcal{L}(\mathcal{D}_1)$, which means that the counterexample the FDFA learner receives is again (a, ab). After the refinement of the FDFA $\mathcal{F}_2$, the FDFA learner now asks an equivalence

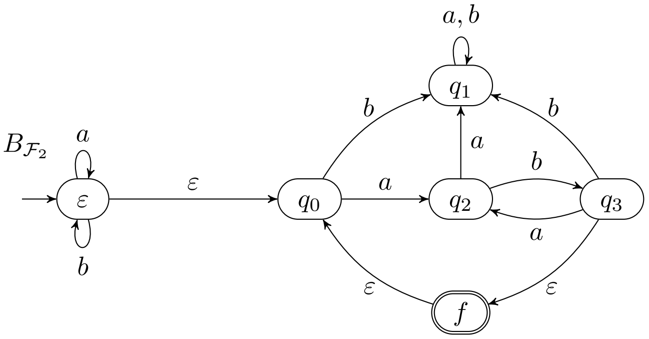


Fig. 22. The BA $B_{\mathcal{F}_2}$ constructed for answering the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_2)$, with $\mathcal{F}_2$ shown in Figure 11

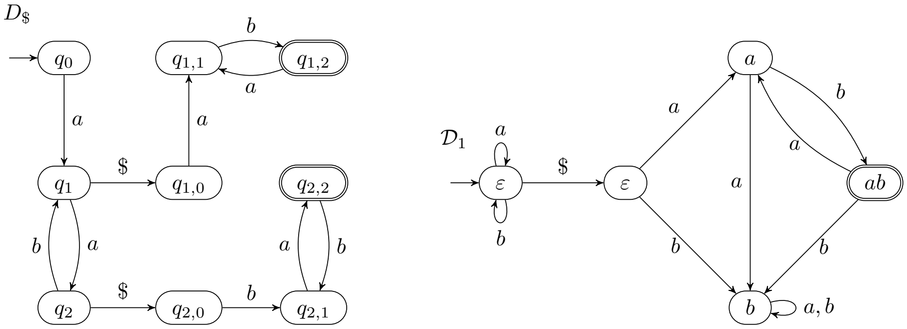


Fig. 23. The DFAs $D_\$$ and $\mathcal{D}_1$ for the FDFA $\mathcal{F}_2$ from Figure 11 and the counterexample (a, ab)

query for the FDFA $\mathcal{F}_3$ shown in Figure 12. On receiving the query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_3)$, the BA $B_{\mathcal{F}_3}$ given in Figure 24 is constructed by the BA learner to ask the BA teacher another equivalence query.

Clearly, $B_{\mathcal{F}_3}$ is not a right conjecture and the BA teacher returns "no" as well as a counterexample, say (bba, ba), which is a negative counterexample for the FDFA teacher. The BA learner builds the two DFAs $D_\$$ and $\mathcal{D}_1$ depicted in Figure 25 in order to get a valid counterexample for the FDFA learner from $\mathcal{L}(D_\$) \cap \mathcal{L}(\mathcal{D}_1)$.

Assume that the FDFA learner receives the counterexample (bb, ab) from the FDFA teacher disguised by the BA learner. The FDFA learner is now able to get the refined FDFA $\mathcal{F}_4$ shown in Figure 13, so he can ask $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_4)$. As usual, the BA learner constructs a BA from $\mathcal{F}_4$ in order to solve the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_4)$ posed by the FDFA learner.

The BA $B_{\mathcal{F}_4}$ constructed by the BA learner is shown in Figure 26 and we can see that $B_{\mathcal{F}_4}$ exactly accepts the target language $L = (ab)^\omega$, which means that $B_{\mathcal{F}_4}$ is already a right conjecture. Therefore, the BA teacher answers positively

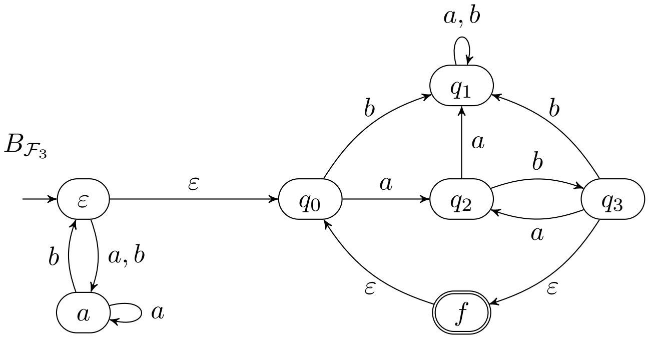


Fig. 24. The BA $B_{\mathcal{F}_3}$ constructed for answering the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_3)$, with $\mathcal{F}_3$ shown in Figure 12

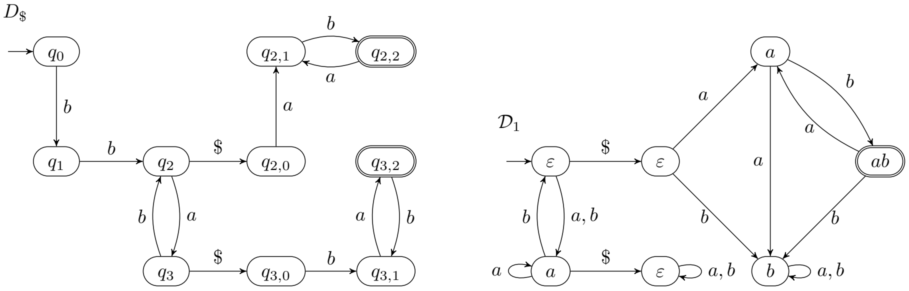


Fig. 25. The DFAs $D_\$$ and $\mathcal{D}_1$ for the FDFA $\mathcal{F}_3$ from Figure 12 and the counterexample (bba, ba)

with "yes" to the BA learner as the result of the equivalence query $\mathrm{EQ}^{\mathrm{BA}}(B_{\mathcal{F}_4})$. The BA learner then outputs the learned BA $B_{\mathcal{F}_4}$ as he has finally completed the learning task.

We notice that the current FDFA $\mathcal{F}_4$ is still not a periodic FDFA of L yet we can build a BA such that $\mathcal{L}(B_{\mathcal{F}_4}) = L$. In practice, the BA learning algorithm very often infers a BA recognizing L before converging to a periodic FDFA of L. In the worst case, the FDFA learner inside the BA learner has to learn a periodic FDFA of L in order to get a right conjectured BA according to Theorem 7.

The BA learner. By means of the previous example, we have introduced informally the ω-regular language learning algorithm, which is formalized in Algorithm 3 as the BA learner. We can note that the learning procedure we described in the running example follows exactly the steps of Algorithm 3. The function *constructBA* is an implementation of the under-approximation BA construction and *ceAnalysis* is the procedure for analyzing counterexamples from the BA teacher. The refinement loop of the conjecture $B_{\mathcal{F}}$ terminates once we get a positive answer from the teacher.

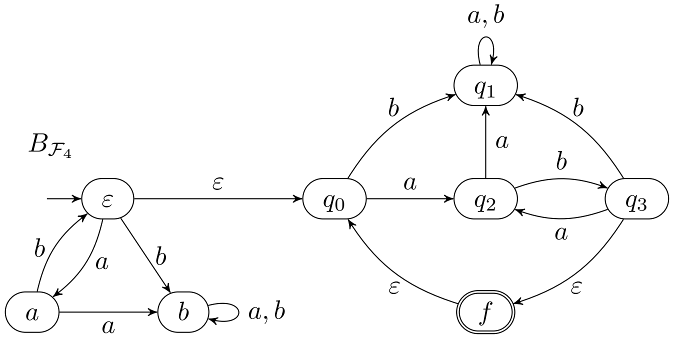


Fig. 26. The BA B_4 constructed for answering the equivalence query $\mathrm{EQ}^{\mathrm{FDFA}}(\mathcal{F}_4)$, with $\mathcal{F}_4$ shown in Figure 13

Algorithm 3: The BA Learner

```
Initialize an FDFA learner $L^\omega$ and get the conjectured FDFA $\mathcal{F}$;
$B_\mathcal{F} = constructBA(\mathcal{F})$;
Let $(a, (u, v))$ be the BA teacher's response on $\mathrm{EQ}^{\mathrm{BA}}(B_\mathcal{F})$;
while $a =$ "no" do
    $(u', v') = ceAnalysis((u, v), \mathcal{F})$;
    Call $L^\omega$ to refine $\mathcal{F}$ with $(u', v')$ and get the new conjectured FDFA $\mathcal{F}$;
    $B_\mathcal{F} = constructBA(\mathcal{F})$;
    Let $(a, (u, v))$ be the BA teacher's response on $\mathrm{EQ}^{\mathrm{BA}}(B_\mathcal{F})$;
return $B_\mathcal{F}$;
```

As discussed before, we can construct from an FDFA $\mathcal{F}$ a BA $B_\mathcal{F}$ such that $\mathrm{UP}(\mathcal{F}) = \mathrm{UP}(\mathcal{L}(B_\mathcal{F}))$ if $\mathcal{F}$ is a periodic FDFA of $\mathrm{UP}(\mathcal{F})$. Thus in the worst case, the FDFA learner inside the BA learner needs to learn a periodic FDFA of target language L in order to get a right conjectured BA. The main result of this section then follows.

Theorem 8 (Correctness and Termination). *The BA learning algorithm based on the FDFA learner and the under-approximation BA construction always terminates and returns a BA recognizing the target ω-regular language L in polynomial time.*

6 Learning to Complement Büchi Automata

As we have seen in Section 3.3, the complementation of Büchi automata [25] is a classic problem that has been extensively studied for more than half a century; see [105] for a survey. The complementation of Büchi automata is a valuable tool in formal verification (cf. [67]), in particular when the property to be satisfied

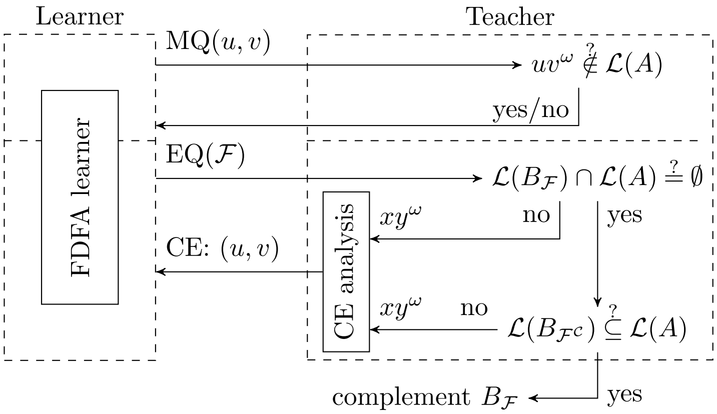

Learner
Teacher
MQ(u, v)
$uv^\omega \overset{?}{\notin} \mathcal{L}(A)$
yes/no
FDFA learner
EQ($\mathcal{F}$)
$\mathcal{L}(B_\mathcal{F}) \cap \mathcal{L}(A) \overset{?}{=} \emptyset$
CE analysis
xy^ω
no
yes
CE: (u, v)
xy^ω
no
$\mathcal{L}(B_{\mathcal{F}^c}) \overset{?}{\subseteq} \mathcal{L}(A)$
complement $B_\mathcal{F}$
yes

Fig. 27. The learning framework for complementing a Büchi automaton A

by a model is given by means of a Büchi automaton, in the program termination analysis (cf. Section 7), and when studying language inclusion problems of ω-regular languages [1, 3, 4]. As Proposition 4 shows, the complementation of Büchi automata is super-exponential, i.e., it can be really expensive in practice as well. While this is generally unavoidable [108], we believe that there is no inherent reason to assume that the complement language is harder than the initial language: in model checking, when the property is given as a formula ϕ, the typical approach assumes that the translation into a Büchi automaton is equally efficient for the formula and its negation, so instead of translating ϕ to A_ϕ and then complementing A_ϕ, it first negates ϕ and then translates $\neg\phi$ to $A_{\neg\phi}$ so that $\mathcal{L}(A_{\neg\phi}) = \Sigma^\omega \setminus \mathcal{L}(A_\phi)$. Would the complement language of ϕ be indeed more complex than the language of ϕ, this approach would suffer in translating the negation of the formula, since such a negation corresponds to the complement of the original property's language. Besides this, we have that complementing twice a language L gives L itself, while complementing a Büchi automaton twice would generate an automaton of incredible size: for instance, complementing twice a BA with 10 states would result in a BA accepting the same language with roughly at least $10^{7 \cdot 10^7}$ states, according to the approximation given by Proposition 4.

This begs to ask the question, whether we can disentangle the complement BA from the syntactic representation of the BA accepting the language we want to complement. By taking inspiration from the regular languages setting, where the minimal DFA accepting a given regular language can be learned by the DFA learning algorithm, in this section we show how we can learn a BA accepting the complement of a given target ω-regular language L.

6.1 The Complement BA Learning Framework

The learning framework for complementing a Büchi automaton is shown in Figure 27 and it has been proposed in [74], to which we refer the interested reader for more details. It is based on a variation of the FDFA learning algorithm to learn $\mathcal{F}$, explained in Sections 4 and 5. As we can see from Figure 27, the learner is exactly the FDFA learner used to learn BAs (cf. Figure 16). This means that the learner first uses membership queries for $\mathcal{F}$ until a consistent automaton is created and then he turns to equivalence queries, while being oblivious of the fact that he is actually learning $\Sigma^\omega \setminus \mathcal{L}(A)$ instead of $\mathcal{L}(A)$. The difference with the learning algorithm for BAs shown in Figure 16 lies completely in the teacher: for membership queries, the teacher uses—cheap—standard queries [11, 73]; the real novelty is in a careful design of the answer to the equivalence queries that makes use of cheap operations whenever possible.

These equivalence queries are *not* executed with the FDFA $\mathcal{F}$ and its complement $\mathcal{F}^\mathcal{C}$, but with the Büchi automata $B_\mathcal{F}$ and $B_{\mathcal{F}^\mathcal{C}}$ that under-approximate them. The teacher first checks whether $\mathcal{L}(B_\mathcal{F})$ is disjoint from $\mathcal{L}(A)$ we want to complement. This step is cheap, and if the answer is negative, then she returns to the learner an ultimately periodic word $uv^\omega \in \mathcal{L}(A)$, where at least some decomposition of uv^ω is (wrongly) accepted by $\mathcal{F}$.

In case $\mathcal{L}(B_\mathcal{F}) \cap \mathcal{L}(A) = \emptyset$, the teacher checks whether the language of $B_{\mathcal{F}^\mathcal{C}}$ is included in the language of A. This is an interesting twist, since language inclusion is one of the traditional justifications for complementing Büchi automata, as mentioned in Section 3.5. But while the problem is PSPACE complete (cf. Proposition 8), it can usually be handled well by using efficient tools like RABIT [1,3,4]. In particular, RABIT makes use of a powerful set of computationally effective preprocessing and automata-exploration based heuristics that usually allow the language inclusion problem to be answered very efficiently.

Non-inclusion comes with a witness in the form of an ultimately periodic word xy^ω accepted by $B_{\mathcal{F}^\mathcal{C}}$, but not by A. Thus, some decomposition (u, v) of xy^ω is (incorrectly) rejected by $\mathcal{F}$ and she returns it to the learner. In case $\mathcal{L}(B_{\mathcal{F}^\mathcal{C}}) \subseteq \mathcal{L}(A)$ holds, the teacher then concludes that $\mathcal{L}(B_\mathcal{F}) = \Sigma^\omega \setminus \mathcal{L}(A)$ and terminates the algorithm with $B_\mathcal{F}$ as the complement of A. Note that the learner is not required to construct an FDFA $\mathcal{F}$ such that $\mathcal{L}(\mathcal{F}) = \Sigma^\omega \setminus \mathcal{L}(A)$; it is enough that $\mathcal{L}(B_\mathcal{F}) = \Sigma^\omega \setminus \mathcal{L}(A)$, which can save the framework to manage further membership and equivalence queries.

More details about the correctness of the proposed complementation framework, its complexity, and its experimental evaluation can be found in [74]. We want, however, to give some more detail about the use of RABIT to solve the language inclusion problem the teacher may need to answer in an equivalence query EQ($\mathcal{F}$). As said above, RABIT is equipped with a powerful set of heuristics; among others, RABIT makes use of the following ones, in an increasing order of their efficacy and amount of computation they need: 1) try simple automata-pruning algorithms, which help in reducing the size of the considered automata; 2) try delayed simulations, which is intended to prove the language inclusion by analyzing the structure of the automata; 3) if inclusion was not established in step 2 then try to find a counterexample to inclusion by the Ramsey-based method [3, 4] with a small timeout value; 4) if no counterexample was found in step 3 then try the automata minimization algorithms proposed in [33], which simplify the two automata by changing their languages while preserving their language inclusion relationship.

Since these heuristics are not complete, RABIT uses as the last resort the Ramsey-based inclusion testing algorithms already used in step 3, this time without timeout, to finally decide whether the language inclusion holds. From the experimental evaluation presented in [74] we can see that the learning-based complementation algorithm is really effective in getting the complement automaton, in particular when the automaton to be complemented is of large size. One interesting thing we noted in the experiments is that the automaton $B_{\mathcal{F}}$ used in the check $\mathcal{L}(B_{\mathcal{F}}) \cap \mathcal{L}(A) = \emptyset$ can change sensibly between an equivalence query and the following one, which makes it difficult to predict how much RABIT is able to exploit its heuristics. Anyway, in very few cases RABIT needed to use the Ramsey-based inclusion testing algorithms to finally decide whether the language inclusion holds, which usually consumes most of the running time in the corresponding experiment.

6.2 The Complement BA Learning Framework in Action

Suppose that we want to learn the complement of the NBA B depicted in Figure 1; recall that $\mathcal{L}(B) = \Sigma^* \cdot b^\omega$. The learning algorithm works as follows: the learner first poses several membership queries and constructs the initial conjectured FDFA $\mathcal{F}_1$ shown in Figure 28.

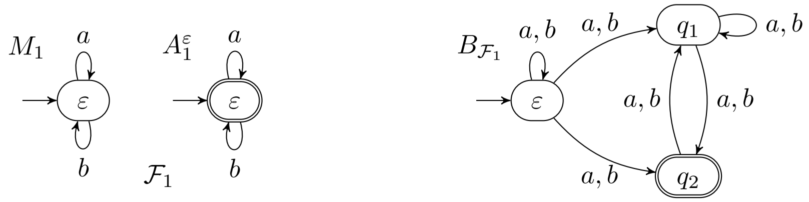


Fig. 28. Initial FDFA $\mathcal{F}_1 = (M_1, \{A_1^\varepsilon\})$ and the corresponding under-approximation Büchi automaton $B_{\mathcal{F}_1}$.

Afterwards, the learner performs the equivalence query $\mathrm{EQ}(\mathcal{F}_1)$ to verify whether $\mathcal{F}_1$ is correct. In order to answer this equivalence query, the teacher first constructs the Büchi automaton $B_{\mathcal{F}_1}$, also shown in Figure 28, and then checks the emptiness of $\mathcal{L}(B_{\mathcal{F}_1}) \cap \mathcal{L}(B)$. This check fails: assume that the teacher gets the ω-word $b(bb)^\omega \in \mathcal{L}(B_{\mathcal{F}_1}) \cap \mathcal{L}(B)$; by means of the counterexample analysis, the teacher is able to answer negatively to the query $\mathrm{EQ}(\mathcal{F}_1)$ posed by the learner by returning the negative counterexample (ε, b), a decomposition of $b(bb)^\omega$.

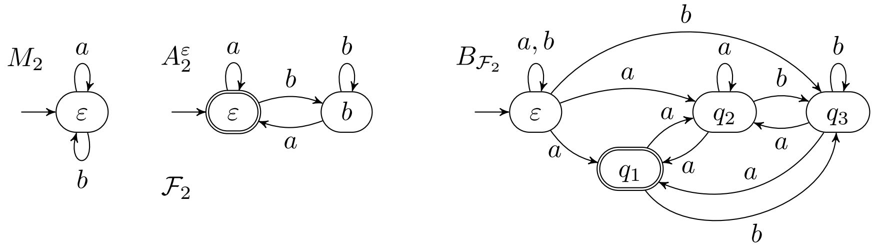


Fig. 29. Second FDFA $\mathcal{F}_2 = (M_2, \{A_2^\varepsilon\})$ and the corresponding under-approximation Büchi automaton $B_{\mathcal{F}_2}$.

Upon receiving (ε, b), the learner refines the current FDFA $\mathcal{F}_1$ to $\mathcal{F}_2$, shown in Figure 29, by means of membership queries; then it poses the equivalence query $\mathrm{EQ}(\mathcal{F}_2)$ for $\mathcal{F}_2$. As before, the teacher first transforms $\mathcal{F}_2$ to $B_{\mathcal{F}_2}$ and then checks for the emptiness of $\mathcal{L}(B_{\mathcal{F}_2}) \cap \mathcal{L}(B)$. It is easy to see that $\mathcal{L}(B_{\mathcal{F}_2})$ is indeed disjoint from $\mathcal{L}(B)$. Therefore, the teacher has first to compute the Büchi automaton $B_{\mathcal{F}_2^\mathcal{C}}$ under-approximating $\mathcal{F}_2^\mathcal{C}$, shown in Figure 30, and then to check the language inclusion $\mathcal{L}(B_{\mathcal{F}_2^\mathcal{C}}) \subseteq \mathcal{L}(B)$; this check fails.

Assume that the teacher finds $b(ab)^\omega \in \mathcal{L}(B_{\mathcal{F}_2^\mathcal{C}}) \setminus \mathcal{L}(B)$; she then answers negatively to $\mathrm{EQ}(\mathcal{F}_2)$ by means of the positive counterexample (b, ab) obtained from $b(ab)^\omega$.

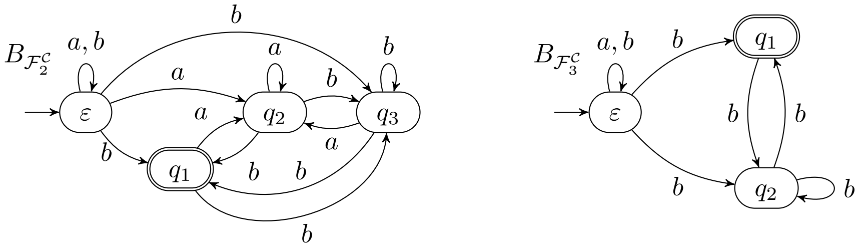


Fig. 30. Under-approximation Büchi automata $B_{\mathcal{F}_2^\mathcal{C}}$ and $B_{\mathcal{F}_3^\mathcal{C}}$ for $\mathcal{F}_2^\mathcal{C}$ (depicted in Figure 29) and $\mathcal{F}_3^\mathcal{C}$ (shown in Figure 31), respectively

The learner uses the received counterexample (b, ab) to further refine the current FDFA $\mathcal{F}_2$; after asking several membership queries, he generates the candidate FDFA $\mathcal{F}_3$ and then asks an equivalence query for it. As in the previous cases, the teacher starts by constructing the Büchi automaton $B_{\mathcal{F}_3}$ for $\mathcal{F}_3$, shown in Figure 31. Since $\mathcal{L}(B_{\mathcal{F}_3}) \cap \mathcal{L}(B)$ is empty, the teacher proceeds to the second check, so she constructs the BA $B_{\mathcal{F}_3^\mathcal{C}}$, shown in Figure 30, and then proceeds to perform the last check, i.e., whether $\mathcal{L}(B_{\mathcal{F}_3^\mathcal{C}}) \subseteq \mathcal{L}(B)$, which is obviously the case. Thus, the teacher terminates the learning algorithm by returning $B_{\mathcal{F}_3}$, shown in Figure 31, as the complement of B.

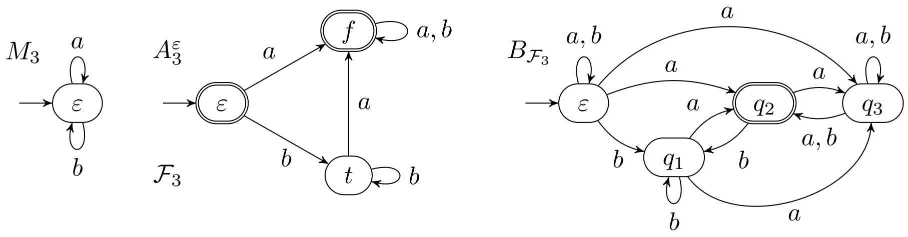


Fig. 31. Final FDFA $\mathcal{F}_3 = (M_3, \{A_3^\varepsilon\})$ and the corresponding under-approximation Büchi automaton $B_{\mathcal{F}_3}$

6.3 Experimental Evaluation

To support our claim that there is no actual super-exponential dependency between the language L we want to complement and the size of the complement $A^{\mathcal{C}}$ of the BA A such that $\mathcal{L}(A) = L$, we briefly recall the experiments we conducted in [74], where the complementation learning framework has been presented.

There we implemented our learning approach as Buechic, based on the ROLL learning library [73]; the inclusion check $\mathcal{L}(B_{\mathcal{F}^{\mathcal{C}}}) \subseteq \mathcal{L}(A)$ (cf. Figure 27) is delegated to RABIT [1, 3, 4]. In the experiments, we compared Buechic with two tools: GOAL [98], which is a mature and well-known tool for manipulating Büchi automata, for which we consider four different implemented complementing algorithms; and SPOT [40], which is the state-of-the-art platform for manipulating ω-automata, including Büchi automata. All tools accept as input automata represented in the Hanoi Omega-Automata (HOA) format [13]. Recall that SPOT does not provide a complementation function for generic Büchi automata directly, thus we first use SPOT to get a deterministic automaton from the given NBA, then complement the resulting deterministic automaton (for parity automata this just means adding 1 to all priorities), and finally transform the resulting complement automaton to an equivalent NBA. Since SPOT is a highly optimized tool that uses effective heuristics, it very often produces very small automata, but the heavy use of heuristics makes the comparison lopsided. Moreover, SPOT uses symbolic data structures called OBDDs which provide a more efficient way to manipulate automata compared to GOAL and Buechic.

Table 1 reports the results of the complementation on the automata from Büchi Store [99], which contains 295 NBAs with 1 to 17 states and with 0 to 123 transitions. However, since one of such automata has only one state without transitions and GOAL fails in recognizing it as a Büchi automaton, we decided to exclude it from the experiments and consider only the remaining 294 cases.

Table 1. Comparison between GOAL, SPOT, and Buechic on complementing Büchi Store. Note that the transitions in SPOT are represented denser—the same automaton attracts a lower transitions count.

Block	Experiments		GOAL				Buechic	SPOT
	(States, Transitions)		Ramsey	Determinization	Rank	Slice		
1	287 NBAs (928, 2071)	$\lvert Q\rvert$	21610	3919	21769	4537	2428	**1629**
		$\lvert\delta\rvert$	964105	87033	179983	125155	35392	**13623**
		t_c	992	300	203	204	105	**6**
2	5 NBAs (55, 304)	$\lvert Q\rvert$	–to–	926	38172	1541	**165**	495
		$\lvert\delta\rvert$		21845	384378	50689	5768	**4263**
		t_c		28	42	12	474	**<1**
3	2 NBAs (20, 80)	$\lvert Q\rvert$	–to–	–to–	27372	11734	**96**	2210
		$\lvert\delta\rvert$			622071	1391424	**6260**	102180
		t_c			56	152	7	**1**

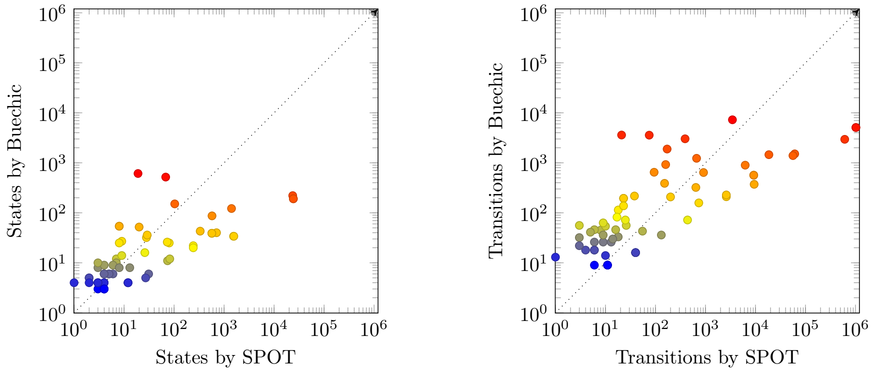


Fig. 32. Comparison between the number of states and transitions of automata generated by SPOT and Buechic on 72 automata corresponding to formulas from [91].

By inspecting the entries in Table 1 we can see that our learning based complementation method always outperforms the complementation methods offered by GOAL when we consider the number of states and transitions. When compared with SPOT, we see that the optimizations in SPOT are really effective, in both runtime and size of generated automata, for the small input automata (cf. block 1), but the transformation to parity automata starts to show its effects for larger automata (cf. blocks 2 and 3).

We have also considered 72 further Büchi automata generated from 72 formulas from [91]. In summary, Ramsey-based, Determinization-based, Rank-based, and Slice-based GOAL approaches solve 49, 58, 61, and 62 complementation tasks, respectively, within 5 minutes, while SPOT solves 66 tasks and Buechic solves 65 tasks. Among these, there are 64 tasks solved by both SPOT and Buechic, while the remaining cases are disjoint, which implies that our algorithm complements existing complementation approaches very well.

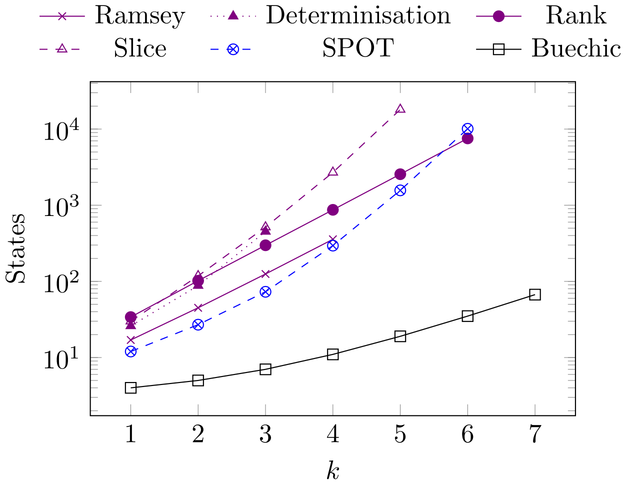


Fig. 33. States comparison of GOAL, SPOT, and Buechic on the formula pattern $\bigwedge_{i=1}^{k}(\mathsf{GF}a_i) \to \mathsf{GF}b$

In Figure 32, relative to the 64 commonly solved tasks, the coordinate values of the y axis and x axis are the corresponding number of states (respectively, transitions) in the complement automata of Buechic and SPOT, respectively. All points below the dotted diagonal indicate that the complement automata learned by our algorithm have smaller values than the complement automata constructed by SPOT, which is the case for almost all large examples. We recall that SPOT merges transitions that share the same source state and target state as one transition, so in the right scatter plot of Figure 32, many points are above the diagonal line. Nevertheless, we can learn from the plots that only SPOT produces those automata with more than 10^3 states or 10^4 transitions, which indicates that the reduction optimizations of SPOT do not work well on large automata and our algorithm performs much better on large automata.

In order to show how the growing trend of the number of states in the complement automata of the complementation algorithms behaves when we increase the size of the given Büchi automata in some cases, we take the generated Büchi automata for the formula pattern $\bigwedge_{i=1}^{k}(\mathsf{GF}a_i) \to \mathsf{GF}b$. The growing trend of the number of states in the complement automata for the approaches in GOAL, SPOT, and Buechic are plotted in Figure 33. The number of states in the complement automaton constructed by GOAL and SPOT is growing exponentially with respect to the parameter k, while the number of states in the complement automaton learned by our algorithm grows linearly. The experimental results show that the performance of our algorithm can be much more stable for some automata with their growth of the states. Thus an advantage of our learning approach is that it has potentially better performance on large automata compared to classic complementation techniques.

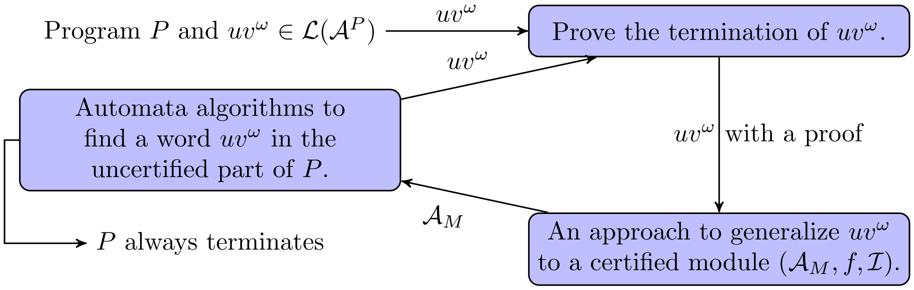

Program P and $uv^\omega \in \mathcal{L}(\mathcal{A}^P)$
uv^ω
Prove the termination of uv^ω.
uv^ω
Automata algorithms to find a word uv^ω in the uncertified part of P.
uv^ω with a proof
$\mathcal{A}_M$
P always terminates
An approach to generalize uv^ω to a certified module $(\mathcal{A}_M, f, \mathcal{I})$.

Fig. 34. The flow of the automata-based termination analysis

7 Application of Büchi Automata in Termination Analysis

In this section we present how Büchi automata and their complements are used in practice for complex verification tasks, like in program termination analysis.

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; see, e.g., [14, 22, 36–38, 48, 49, 53, 55, 64, 69, 70, 80, 84–86, 94, 100, 101]. In general, while analyzing the termination of a program, we need to deal with the following challenge: when a program contains loops with branching or nesting, how can we devise a termination argument that holds for any possible interleaving of the 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 for this class of programs can be done rather efficiently [15–17, 23, 35, 54, 71, 83], but its extension to general programs is not easy.

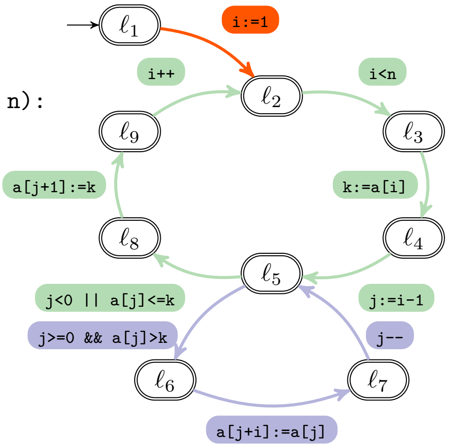
7.1 Automata-Based Termination Analysis

In order to simplify our presentation, we consider only C programs without function calls and pointers; the variable updates are restricted to linear combinations. Since our goal in this section is to describe the modular termination analysis for a given program P, we assume that every sampled path can be proved to be terminating. Therefore, in the end, we can prove that P always terminates.

The approach of Heizmann *et al.* [55] proposes a modular construction of termination proofs for a general program P from termination proofs of lasso-shaped programs obtained from its concrete paths as depicted in Figure 34. On a high level, the approach repeatedly performs the following sequence of operations: first, it samples a path $\tau = uv^\omega$ from the possible behaviours of P and attempts

```
program insertionSort(int a[], int n):
ℓ1: int i:=1
ℓ2: while (i<n)
ℓ3:   int k:=a[i]
ℓ4:   int j:=i-1
ℓ5:   while (j>=0 && a[j]>k):
ℓ6:     a[j+1]:=a[j]
ℓ7:     j--
ℓ8:   a[j+1]:=k
ℓ9:   i++
```

(a) Program $P^{\texttt{ins}}$



(b) The BA $\mathcal{A}^{P^{\texttt{ins}}}$

Fig. 35. An example of program and its BA representation

to prove its termination [15–17,23,35,54,71,83] by using an off-the-shelf termination checker, like LassoRanker, part of the Ultimate Automizer suite [55]. The returned result of this step is possibly a termination argument of the sampled path, a non-termination argument of the sampled path, or "unknown" which indicates that the termination checker failed to decide the termination of the sampled path. 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 behaviour of P contains a path τ' not covered by any certified module generated so far and, if so, the procedure is repeated. This sequence is repeated until either a non-terminating path is found, "unknown" is returned, or all behaviours of P are covered by the modules.

7.2 Automata-Based Termination Analysis: an Example

As an example of the above approach, consider the insertion sort program $P^{\texttt{ins}}$ shown in Figure 35(a); Figure 35(b) shows the control flow graph (CFG) of $P^{\texttt{ins}}$ as a Büchi automaton $\mathcal{A}^{P^{\texttt{ins}}}$.

The alphabet of $\mathcal{A}^{P^{\texttt{ins}}}$ is the set of all statements occurring in $P^{\texttt{ins}}$, like assignments and guards, while the states of $\mathcal{A}^{P^{\texttt{ins}}}$ are the locations of $P^{\texttt{ins}}$; the initial state is the first location of the program, i.e., its entry point. The transitions connect states according to the way each location is reachable from another: for instance, we have the transition from ℓ_1 to ℓ_2 with action `i:=1` since ℓ_2 is reached after such initialization in location ℓ_1; similarly, we have a transition from ℓ_5 to ℓ_8 with action `j<0 || a[j]<=k` since ℓ_8 is reached when the guard `j>=0 && a[j]>k` of the `while` statement at location ℓ_5 is not satisfied. All states of $\mathcal{A}^{P^{\text{ins}}}$ are accepting so each feasible infinite sequence of statements of the program corresponds to an infinite word in the language $\mathcal{L}(\mathcal{A}^{P^{\text{ins}}})$.

The aim of the termination analysis is to cover the executions of $\mathcal{A}^{P^{\text{ins}}}$ by the accepted words of a finite set of BAs $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ such that $\mathcal{L}(\mathcal{A}^{P^{\text{ins}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \cdots \cup \mathcal{L}(\mathcal{A}_n)$ which is reduced to checking whether $\mathcal{L}(\mathcal{A}^{P^{\text{ins}}}) \cap \mathcal{L}(\mathcal{A}_1^{\mathcal{C}}) \cap \cdots \cap \mathcal{L}(\mathcal{A}_n^{\mathcal{C}}) = \emptyset$, as we have seen in Section 3.5. If we can prove that each BA $\mathcal{A}_i$ represents a program with a termination argument, then since every execution of P^{ins} is represented by a word in $\mathcal{A}^{P^{\text{ins}}}$, P^{ins} is guaranteed to terminate by the arguments for the single BAs $\mathcal{A}_i$.

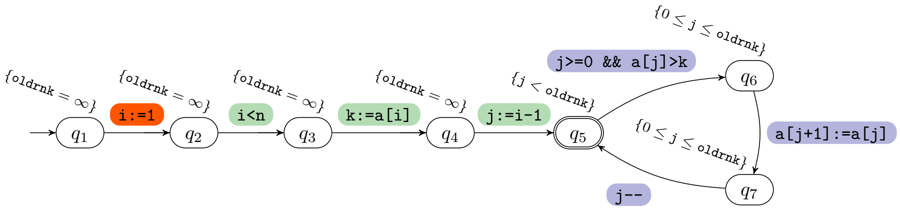


Fig. 36. A certified module for the lasso word $uv^{\omega} =$ `i:=1` $\cdot$ `i<n` $\cdot$ `k:=a[i]` $\cdot$ `j:=i-1` $\cdot$ (`j>=0 && a[j]>k` $\cdot$ `a[j+i]:=a[j]` $\cdot$ `j--` $)^{\omega}$

In order to have a termination argument, 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. The triple $\mathcal{M}_i = (\mathcal{A}_i, f_i, \mathcal{I}_i)$ is called a *certified module*. The construction of the set $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ is progressive (see Figure 34). First, we sample an ultimately periodic word $uv^{\omega} \in \mathcal{L}(\mathcal{A}^{P^{\text{ins}}})$—which is essentially a lasso-shaped program—and use an off-the-shelf tool to check if it corresponds to a terminating argument. In our example, we start with sampling the word $uv^{\omega} =$ `i:=1` $\cdot$ `i<n` $\cdot$ `k:=a[i]` $\cdot$ `j:=i-1` $\cdot$(`j>=0 && a[j]>k` $\cdot$ `a[j+i]:=a[j]` $\cdot$ `j--` $)^{\omega}$. We can prove termination of the path corresponding to uv^{ω} by finding, e.g., the ranking function $f_1(\mathtt{i}, \mathtt{j}, \mathtt{n}) = \mathtt{j} + 1$, for which it holds that at each iteration of the inner loop, the value of $f_1(\mathtt{i}, \mathtt{j}, \mathtt{n})$ decreases since `j` is decreased by 1. The resulting certified module is shown in Figure 36, where `oldrnk` is a fresh variable used to keep track of the value of the ranking function at the previous visit of the accepting state.

In the following, we denote the inner loop of $\mathcal{A}^{P^{\text{ins}}}$ as $\textsc{Inner} =$ `j>=0 && a[j]>k` $\cdot$ `a[j+i]:=a[j]` $\cdot$ `j--` and its outer loop as $\textsc{Outer} =$ `j<0 || a[j]<=k` $\cdot$ `a[j+1]:=k` $\cdot$ `i++` $\cdot$ `i<n` $\cdot$ `k:=a[i]` $\cdot$ `j:=i-1`. 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 $\mathcal{L}_1 =$ `i:=1` $\cdot$ `i<n` $\cdot$ `k:=a[i]` $\cdot$ `j:=i-1` $\cdot(\textsc{Inner}+\textsc{Outer})^{*} \cdot \textsc{Inner}^{\omega}$. 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 $\mathcal{M}_1 = (\mathcal{A}_1, f_1, \mathcal{I}_1)$, depicted in Figure 37; the transitions labelled with action $\textsc{Inner}$ or $\textsc{Outer}$ are a shorthand for the corresponding

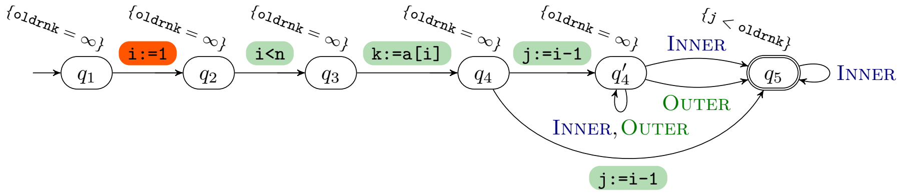


Fig. 37. The certified module $\mathcal{M}_1$ for the language $\mathcal{L}_1 = \texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1}(\textsc{Inner} + \textsc{Outer})^* \cdot \textsc{Inner}^\omega$

sequences of transitions and states: for instance, the self-loop on q_5 with action $\textsc{Inner}$ stands for the states and transitions reachable from q_5 in Figure 36.

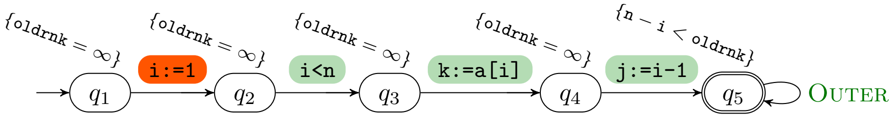


Fig. 38. A certified module for the lasso word $\texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1} \cdot \textsc{Outer}^\omega$

We proceed by removing all paths covered by $\mathcal{L}_1$ from $\mathcal{A}^{P^{\text{ins}}}$ to know which paths still need to be examined. The removal can be performed by executing a *BA difference algorithm*, presented in Section 3.4, 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 $\mathcal{L}(\mathcal{A}^{P^{\text{ins}}}_{|\mathcal{A}_1}) = \texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1} \cdot (\textsc{Inner}^* \cdot \textsc{Outer})^\omega$ represented by $\mathcal{A}^{P^{\text{ins}}}_{|\mathcal{A}_1}$. Suppose the next sampling gives us $uv^\omega = \texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1} \cdot \textsc{Outer}^\omega$, for which, e.g., the ranking function $f_2(\mathtt{i}, \mathtt{j}, \mathtt{n}) = \mathtt{n} - \mathtt{i}$ is applicable; the corresponding certified module is shown in Figure 38.

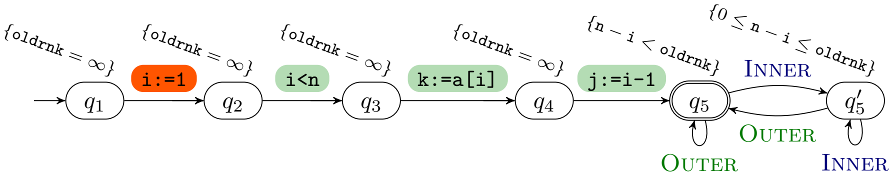


Fig. 39. The certified module $\mathcal{M}_2$ for the language $\mathcal{L}_2 = \texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1} \cdot (\textsc{Inner}^* \cdot \textsc{Outer})^\omega$

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 $\mathcal{L}_2 = \texttt{i:=1} \cdot \texttt{i<n} \cdot \texttt{k:=a[i]} \cdot \texttt{j:=i-1} \cdot (\textsc{Inner}^* \cdot \textsc{Outer})^\omega$. We represent these paths by the certified module $\mathcal{M}_2 = (\mathcal{A}_2, f_2, \mathcal{I}_2)$ where $\mathcal{L}(\mathcal{A}_2) = \mathcal{L}_2$, shown in Figure 39.

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

7.3 Automata-Based Termination Analysis: Difficulties

As we have seen in the example above, the general termination analysis involves several operations based on Büchi automata, like emptiness check and complementation or language difference. While emptiness is really cheap (cf. Proposition 6), complementation or language difference are in general extremely expensive (cf. Propositions 4 and 5), so it would be better to limit their number as much as possible.

Note however that the complementation of a lasso-shaped automaton corresponding to a lasso-shaped word is really easy: it is enough to add an accepting sink state collecting all missing transitions and make the original accepting state no more accepting. While this is cheap, the net effect in the termination analysis is really limited: in this way we remove only one infinite word at a time, so there is a negligible progress in the termination analysis.

The generalizations we have seen before are useful to avoid such negligible progress, since they allow us to remove a possibly very large set of words at each iteration. This comes at the expense of the complexity of complementing the corresponding Büchi automaton, which now can suffer from the super-exponential complexity of the language difference or complementation operations.

It is easy to recognize that there is a trade-off between the size of the set of paths of the program P covered by current certified module $\mathcal{M}_i$ and the complexity of complementing $\mathcal{M}_i$ itself. There are several techniques to balance these two aspects, together with specialized algorithms for them; see [28] for more details on the different generalization techniques, their effectiveness in covering the paths of the input program, and further explanations and references about the creation of certified modules from a lasso-shaped word.

We are confident that learning the complement of Büchi automata, shown in Section 6, is a useful technique that can complement the existing proposals, in particular for tackling the more challenging cases where the ordinary techniques start suffering from the super-exponential grown of the complement BA.

8 Conclusion

In this work, we have presented a learning algorithm for Büchi automata by means of its learning of the simple ω-regular language $(ab)^\omega$. We have also demonstrated how the learning algorithm can be used in classical automata operations such as complementation checking and in the termination analysis context. We believe that with the intuitive explanation of the different learning algorithms for both finite and ω-regular languages, it will benefit both the learning community and the model checking community.

References

1. RABIT tool. http://languageinclusion.org/doku.php?id=tools.
2. F. Aarts and F. W. Vaandrager. Learning I/O automata. In *CONCUR*, pages 71–85, 2010.
3. P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In *CAV*, volume 6174, pages 132–147, 2010.
4. P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In *CONCUR*, volume 6901, pages 187–202, 2011.
5. J. D. Allred and U. Ultes-Nitsche. A simple and optimal complementation algorithm for Büchi automata. In *LICS*, pages 46–55, 2018.
6. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.
7. R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.
8. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
9. D. Angluin, U. Boker, and D. Fisman. Families of DFAs as acceptors of omega-regular languages. In *MFCS*, pages 11:1–11:14, 2016.
10. D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *IJCAI*, pages 3308–3314, 2015.
11. D. Angluin and D. Fisman. Learning regular omega languages. *Theoretical Computer Science*, 650:57–72, 2016.
12. A. Arnold. A syntactic congruence for rational ω-languages. *Theoretical Computer Science*, 39:333–335, 1985.
13. T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Kretínský, D. Müller, D. Parker, and J. Strejcek. The Hanoi omega-automata format. In *CAV*, pages 479–486, 2015.
14. A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. *Logical Methods in Computer Science*, 6, 2010.
15. A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, pages 51–62, New York, NY, USA, 2013. ACM.
16. A. M. Ben-Amram and S. Genaim. Complexity of Bradley-Manna-Sipma lexicographic ranking functions. In *CAV*, volume 9207, pages 304–321. Springer, 2015.
17. A. M. Ben-Amram and S. Genaim. On multiphase-linear ranking functions. In *CAV*, volume 10427, pages 601–620. Springer, 2017.
18. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.
19. F. Blahoudek, M. Heizmann, S. Schewe, J. Strejček, and M.-H. Tsai. Complementing semi-deterministic Büchi automata. In *TACAS*, pages 770–787, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

20. B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI*, pages 1004–1009, 2009.
21. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *CAV*, pages 360–364, 2010.
22. C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination through conditional termination. In *TACAS*, volume 10205, pages 99–117, Berlin, Heidelberg, 2017. Springer.
23. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, Berlin, Heidelberg, 2005. Springer-Verlag.
24. S. Breuers, C. Löding, and J. Olschewski. Improved Ramsey-based Büchi complementation. In *FOSSACS*, pages 150–164, 2012.
25. J. R. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, 1962.
26. H. Calbrix, M. Nivat, and A. Podelski. Ultimately periodic words of rational ω-languages. In *MFPS*, pages 554–566, 1993.
27. M. Chapman, H. Chockler, P. Kesseli, D. Kroening, O. Strichman, and M. Tautschnig. Learning the Language of Error. In *ATVA*, pages 114–130, 2015.
28. Y.-F. Chen, M. Heizmann, O. Lengál, Y. Li, M.-H. Tsai, A. Turrini, and L. Zhang. Advanced automata-based algorithms for program termination checking. In *PLDI*, pages 135–150, 2018.
29. Y.-F. Chen, C. Hsieh, O. Lengál, T.-J. Lii, M.-H. Tsai, B.-Y. Wang, and F. Wang. PAC Learning-based Verification and Model Synthesis. In *ICSE*, pages 714–724, 2016.
30. Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8(2):117–141, 1974.
31. E. M. Clarke. Model checking - my 27-year quest to overcome the state explosion problem. In *LPAR*, page 182, 2008.
32. E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
33. L. Clemente and R. Mayr. Advanced automata minimization. In *Proceedings of POPL'13*, pages 63–74. ACM, 2013.
34. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.
35. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.
36. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, New York, NY, USA, 2006. ACM.
37. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.
38. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, pages 47–61, Berlin, Heidelberg, 2013. Springer-Verlag.
39. J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *USENIX*, pages 193–206, 2015.
40. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, pages 122–129, 2016.
41. E. A. Emerson and C. Lei. Modalities for model checking: Branching time strikes back. In *POPL*, pages 84–96, 1985.

42. E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.
43. A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS*, pages 2–17, 2008.
44. L. Feng, M. Kwiatkowska, and D. Parker. Compositional verification of probabilistic systems using learning. In *QEST*, pages 133–142, 2010.
45. S. Fogarty, O. Kupferman, M. Y. Vardi, and T. Wilke. Profile trees for Büchi word automata, with application to determinization. *Information and Computation*, 245:136–151, 2015.
46. S. Fogarty, O. Kupferman, T. Wilke, and M. Y. Vardi. Unifying Büchi complementation constructions. *Logical Methods in Computer Science*, 9(1), 2013.
47. E. Friedgut, O. Kupferman, and M. Y. Vardi. Büchi complementation made tighter. In *ATVA*, pages 64–78, 2004.
48. P. Ganty and S. Genaim. Proving termination starting from the end. In *CAV*, volume 8044, pages 397–412, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
49. J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58:3–31, 2017.
50. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.
51. O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010.
52. S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *CHARME*, pages 96–110, 2003.
53. W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *SAS*, pages 304–319, Berlin, Heidelberg, 2010. Springer-Verlag.
54. M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In *ATVA*, volume 8172, pages 365–380, Cham, 2013. Springer.
55. M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV*, pages 797–813. Springer-Verlag New York, Inc., 2014.
56. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., 2006.
57. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring Canonical Register Automata. In *VMCAI*, pages 251–266, 2012.
58. M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1-2):65–98, 2014.
59. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.
60. M. Isberner, F. Howar, and B. Steffen. The open-source LearnLib. In *CAV*, pages 487–495, 2015.
61. D. Kähler and T. Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In *ICALP*, pages 724–735, 2008.
62. M. Kaminski. A classification of ω-regular languages. *Theoretical Computer Science*, 36:217–229, 1985.

63. M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA, 1994.
64. D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, pages 89–103, Berlin, Heidelberg, 2010. Springer-Verlag.
65. O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Transaction on Computer Logic*, 2(3):408–429, 2001.
66. R. P. Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Sciences*, 35(1):59–71, 1987.
67. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.
68. L. H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3(4):376–384, 1969.
69. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *PLDI*, pages 489–498, New York, NY, USA, 2015. ACM.
70. W. Lee, B. Wang, and K. Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, Berlin, Heidelberg, 2012. Springer-Verlag.
71. J. Leike and M. Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1), 2015.
72. Y. Li, Y. Chen, L. Zhang, and D. Liu. A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. *CoRR*, abs/1610.07380, 2016.
73. Y. Li, Y. Chen, L. Zhang, and D. Liu. A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. In *TACAS*, pages 208–226, 2017.
74. Y. Li, A. Turrini, L. Zhang, and S. Schewe. Learning to complement Büchi automata. In *VMCAI*, volume 10747, pages 313–335, 2018.
75. S.-W. Lin, E. André, Y. Liu, J. Sun, and J. S. Dong. Learning assumptions for compositional verification of timed systems. *IEEE Transactions on Software Engineering*, 40(2):137–153, 2014.
76. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.
77. O. Maler and L. Staiger. On syntactic congruences for omega-languages. In *STACS*, pages 586–594, 1993.
78. K. L. McMillan. *Symbolic model checking.* Kluwer, 1993.
79. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *POPL*, pages 613–625, 2017.
80. O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham. Reducing liveness to safety in first-order logic. *ACM on Programming Languages*, 2(POPL):26:1–26:33, 2018.
81. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.
82. N. Piterman. From nondeterministic Büchi and streett automata to deterministic parity automata. In *LICS*, pages 255–264, 2006.
83. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, volume 2937, pages 239–251, Berlin, Heidelberg, 2004. Springer.
84. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41, Washington, DC, USA, 2004. IEEE Computer Society.
85. A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, pages 314–327, Berlin, Heidelberg, 2008. Springer-Verlag.

86. C. Popeea and A. Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS*, pages 237–251, Berlin, Heidelberg, 2012. Springer-Verlag.
87. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC*, pages 411–420, 1989.
88. S. Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327, 1988.
89. S. Schewe. Büchi complementation made tight. In *STACS*, volume 3 of *LIPIcs*, pages 661–672, 2009.
90. S. Schewe. Tighter bounds for the determinisation of Büchi automata. In *FOSSACS*, pages 167–181, 2009.
91. S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-deterministic Büchi automata for linear temporal logic. In *CAV*, pages 312–332, 2016.
92. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with appplications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
93. L. Staiger. Research in the theory of omega-languages. *Elektronische Informationsverarbeitung und Kybernetik*, 23(8/9):415–439, 1987.
94. T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58:33–65, 2017.
95. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Sematics, chapter 4, pages 133–192. 1990.
96. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 389–455. 1997.
97. M. Tsai, S. Fogarty, M. Y. Vardi, and Y. Tsay. State of Büchi complementation. *Logical Methods in Computer Science*, 10(4), 2014.
98. M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for games, omega-automata, and logics. In *CAV*, pages 883–889, 2013.
99. Y.-K. Tsay, M.-H. Tsai, J.-S. Chang, and Y.-W. Chang. Büchi store: An open repository of Büchi automata. In *TACAS*, pages 262–266, 2011.
100. C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS*, pages 54–70, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
101. C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. In *ESOP*, pages 412–431, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
102. F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, Jan. 2017.
103. G. van Heerdt, M. Sammartino, and A. Silva. CALF: categorical automata learning framework. In *CSL*, pages 29:1–29:24, 2017.
104. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata*, pages 238–266, 1995.
105. M. Y. Vardi. The Büchi complementation saga. In *STACS*, volume 4393, pages 12–22, 2007.
106. M. Y. Vardi and T. Wilke. Automata: from logics to algorithms. In *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas].*, pages 629–736, 2008.
107. F. Wang, J. Wu, C. Huang, and K. Chang. Evolving a Test Oracle in Black-Box Testing. In *FASE*, pages 310–325, 2011.

108. Q. Yan. Lower bounds for complementation of ω-automata via the full automata technique. *Logical Methods in Computer Science*, 4(1:5), 2008.